université
PARIS-SACLAY

Interoperability between proof systems using the logical framework Dedukti)

Thèse de doctorat de l'Université Paris-Saclay

École doctorale n° 580, Sciences et technologies de l'information et de la communication (STIC)
Spécialité de doctorat: Informatique
Unité de recherche: UMR9021
Référent: ENS Paris-Saclay

Thèse présentée et soutenue à Gentilly, le 3 décembre 2020, par

François Thiré

Composition du jury:

Roberto Di Cosmo Professeur, Université Paris Diderot	Président/e
Cézary Kaliszyk Professeur, University of Innsbruck	Rapporteur
Herman Geuvers Professeur, Faculty of Science Nijmegen	Rapporteur
Chantal Keller Maître de conférence, Université Paris-Saclay	Examinatrice
Raphaël Cauderlier Ingénieur de recherche, Nomadic Labs	Examinateur
Véronique Benzaken Professeur, Université Paris-Sud 11	Examinatrice
Gilles Dowek Directeur de recherche, Inria	Directeur
Stéphane Graham-Lengrand Professeur, SRI International	Codirecteur

Thèse de doctorat

NNT: 2020UPASG053

Contents

-

I

Meta-theory of Cumulative Types Systems and their embeddings to the $\lambda\Pi$-calculus modulo theory

Introduction

Foundations of Mathematics

On which basis are mathematics founded? What is the common knowledge that is used to prove a mathematical statement? Today, this question has many answers and this thesis is an approach among many others to gather these different answers.

The historical answer was to use natural languages as the language of mathematics. The problem with natural languages is that they are by essence ambiguous. A famous example written in *English* is given by the sentence *I saw a man on a hill with a telescope* which has several interpretations:

- There is a man on a hill, and I am seeing him with a telescope
- There is a man on a hill, who I am seeing, and he has a telescope
- There is a man on a hill which has also a telescope on it
- I am on a hill, I saw a man using a telescope
- There is a man on a hill, he is using a telescope and I am seeing him

So even if natural languages have been used for centuries to write mathematics, mathematicians and philosophers were trying to find a better framework to express mathematics to avoid any ambiguity and to be sure that proofs are valid. Such a framework is called a *formal system*.

The German philosopher Gottlob Frege is one of the first to provide a concrete solution with Predicate Logic[1] [Fre93]. Predicate Logic provides at the same time both a language to express mathematical statements (also called *propositions*) and a system to prove these statements. For example, one may write in Predicate Logic sentences such as: $\forall x, x = x$ (the reflexivity of equality) or $\forall x, \forall y, \forall z, (x = y) \Rightarrow (y = z) \Rightarrow x = z$ (transitivity of equality). In this context, x is a *variable* but also a mathematical expression, $=$ is called a *predicate*, it builds a proposition from mathematical expressions, $\Rightarrow$ is called a connective, it constructs a proposition from other propositions, and $\forall$ is called a *quantifier*, it *binds* a variable to a proposition. However it cannot be used as it is to provide a foundation for mathematics because Predicate Logic does not know about *mathematical objects* such as natural numbers or functions. Hence, Predicate Logic needs a *theory*, a set of mathematical statements–called *axioms*–which are considered to be true to prove theorems about mathematical objects.

[1] The current version of Predicate Logic is due to Ackerman & Hilbert but the ideas go back to Frege

To sum up

Mathematics should be written in a *formal system*. A language to write and prove mathematical statements also called *propositions*. To be effectively used, Predicate Logic needs a theory, a set of mathematical statements called axioms which are assumed to be true.

Elementary Set Theory

A first proposition for a theory in Predicate Logic to express mathematical knowledge is what is called today *elementary set theory* [Can74] [Hal17]. The main idea of this theory is that any mathematical object can be constructed from one fundamental object: *sets*. This theory describes what a set is and how new sets can be constructed. For example, the theory has an axiom which postulates that there exists an empty set, a set which contains nothing ; it has another axiom which postulates that if there are two sets a and b then there is a set which contains a and b (*axiom of pairing*). However, the first formulation of this theory could build exotic sets such as *the set of all sets*. A direct consequence of such construction is that this set contains itself. This weirdness leads to a *logical inconsistency* (a paradox) meaning that every proposition could be proven in this theory such as $2 + 2 = 5$. Of course, a good foundation for mathematics should avoid such logical inconsistency. This paradox has been discovered by the logician Bertrand Russell [Irv95] which could be summed up as follow: *I lie*. If such a sentence is allowed to be formed, which is the case in English, this leads to a paradox: If this sentence is true, then it is also false and vice versa. This paradox created a *schism* in the foundations of mathematics: On one side, people have been trying to fix the original theory by modifying the axioms of elementary set theory ; this led to other theories such as **ZF** (set theory) [Jec13] or **NBG** (theory of classes) [God25] [God28]. Predicate Logic with **ZFC**, an extension of **ZF** with the *axiom of choice* is an answer to the foundation of mathematics used by most mathematicians today. On the other side people were trying to change radically the set of axioms to propose a new foundation for mathematics called *type theory*.

To sum up

Set Theory is a theory expressed in Predicate Logic. Most of our mathematical knowledge today is expressed in this theory.

Type theory

A first description of *type theory* was made in a book called *Principia Mathematica* [WR27] written by Bertrand Russell and Alfred North Whitehead which proposes to build mathematics upon a new theory called *type theory*. The main idea behind type theory is that any *term* meaning any mathematical expression (such as $2 + 2$, 4, $+$, $x \mapsto x + 2$) has a type. For example, the *type* of $2 + 2$ and 4 is a natural number (denoted $\mathbb{N}$). To relate a term and its type we generally use a semi-colon as in $2 + 2 : \mathbb{N}$ or $4 : \mathbb{N}$. $+$ is an operation which expects two natural numbers and returns a natural number. Its type is denoted $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$. The expression $x \mapsto x + 2$ takes a natural number x and returns the natural number $x + 2$. Its type is denoted $\mathbb{N} \to \mathbb{N}$. Typing rules act as semantic rules for English. In English, a sentence such as "*I eat rain*" is gramatically

correct but conveys no meaning. In type theory, types rule out ill-formed mathematical terms such as $2 + (x \mapsto +2) = 4$.

Type theory has many variants just like set theory, even if only one type theory was formulated in *Principia Mathematica*. These theories have proliferated during the second half of the twentieth century after the work of Alonzo Church on λ-calculus.

To sum up

Type theory provides an alternative to Predicate Logic where every mathematical expression have a type. Such type is used to avoid paradoxes.

λ-calculus

Type theory gained interest with the work of Alonzo Church on λ-calculus [Chu36]. λ-calculus can be seen as an alternative for set theory where *functions* are the fundamental objects. The name comes from the symbol λ which is used to introduce functions: Hence, instead of denoting the function $x \mapsto x + x$, in λ-calculus this function is denoted $\lambda x.\ x + x$. Using functions as a primitive element[2] brings something very powerful which did not exist before in Predicate Logic with set theory: *computation*. To emphasize this change of perspective, let us take an example. In set theory, one may prove that $2 + 2 = 4$ using Peano axioms [Pea89] and the transitivity of equality:

$$\dfrac{2 + 2 = 3 + 1 \qquad \dfrac{3 + 1 = 4 + 0 \qquad 4 + 0 = 4}{3 + 1 = 4}}{2 + 2 = 4}$$

In Alonzo Church λ-calculus, $+$ can be defined as a computable function. Therefore $2 + 2$ *computes* to 4. Hence, a proof of $2 + 2 = 4$ in Alonzo Church theory is a direct consequence of the reflexivity of equality:

$$\dfrac{4 = 4}{2 + 2 = 4}$$

A direct consequence of this is that proofs are much shorter.

To sum up

λ-calculus is an alternative to set theory. The notion of *computation* is at the heart of this theory.

However, functions are not enough to build a foundation for mathematics because in Alonzo Church's λ-calculus, it is possible to write functions whose computation does not terminate. Such functions are not suitable for mathematical foundations because they lead directly to logical inconsistencies. This is why Alonzo Church proposed to add *types* on top of the λ-calculus. With type theory, bad functions are considered *ill-formed* and are ruled out by the typing rules.

[2] in set theory, functions are just sets

The combination of functions as primitive elements and type systems are the basis of many type systems that exist today. The one formulated by Alonzo Church is called Simple Type Theory [Chu40]. Besides functions in Simple Type Theory, there is one connective $\Rightarrow$ and one quantifier $\forall$. Functions are introduced with the symbol λ which is also a binder. In Simple Type Theory, axioms are often presented as derivation rules. A derivation rule relates *judgments*. A judgment is built from a context (often denoted Γ) and a proposition as in $\Gamma, P \vdash Q$. An example of derivation rule is given below

$$\frac{\Gamma, P \vdash Q}{\Gamma \vdash P \Rightarrow Q}$$

which means that if I can deduce the proposition Q assuming that the context Γ and the proposition P are true, then I can deduce in the context Γ the proposition $P \Rightarrow Q$ is true. The judgment (or judgments) above the line are called *premises* and the one below is called *conclusion*. By chaining rules of this kind, we may construct proofs. A conclusion may be used as the premise of another rule. This chaining of rules gives a tree and such proof is called a *derivation tree*. For example, one may derive a proof of the proposition $\forall P, P \Rightarrow P$ in Simple Type Theory as follows:

$$\frac{\dfrac{P \vdash P}{\vdash P \Rightarrow P}}{\vdash \forall P, P \Rightarrow P}$$

Such derivation rules are also used to describe the typing rules of a system. In this case, a derivation tree is also called a *typing derivation*.

To sum up

λ-calculus can be formulated outside Predicate Logic using type theory.

Proposition-as-type principle

A central idea in type theory which appears years later after Church's λ-calculus is that a type can be used to express a proposition. Instead of having two sets of derivation rules, one for the typing rules, and one for the proof system, there is only one for the typing rules. Hence a proof becomes a term and in the case of the λ-calculus is also most of the time a function. Using this principle, a proof of a theorem is valid if the type of the proof (the function) is the theorem itself. This idea, which is often called Curry-Howard correspondence in honor to the mathematicians Haskell Curry and William Howard [Cur34] [How80] has been very fruitful and extended in many ways in computer science. However, to be effective, type systems which use this principle also introduce a notion of computation inside the types. For example, in such system, $4 = 4$ and $2 + 2 = 4$ are types and (as explained before) the same type because $2 + 2$ computes to 4. This notion of computation for types leads to another equality $\equiv$ which is called *judgmental* equality or *denotational* equality. For example $(2 + 2 = 4) \equiv (4 = 4)$.

This is why in type theory we have two kind of equalities:

- A propositional equality as in $2 + 2 = 4$
- A judgmental (or computational) equality as in $(2 + 2 = 4) \equiv (4 = 4)$

Actually, the equality $(2 + 2 = 4) \equiv (4 = 4)$ comes from $2 + 2 \equiv 4$. From the reflexivity of propositional equality, we obtain that every time $a \equiv b$ then $a = b$. The other implication is generally not true. One reason is that the other implication changes the logic itself and therefore can be seen as an axiom (or rule) for the theory. Type theories which have this axiom are generally called *extensional* (in opposition to *intentional* type theory when the axiom is not valid).

From this argument, we see that the status of what is qualified as an *axiom* changes a little when it comes to computation in type theory using the proposition-as-type principle. We have *non-computable axioms* as in set theory which does not change the computational equality $\equiv$ and *computable axiom* which enriches this equality. Moreover, because we use the proposition-as-type principle, these axioms are given directly by the typing rules themselves.

The proposition-as-type principle introduces a parallel with programming languages where programs also have a type. In that case, a program can be considered as proof that the type is inhabited. This parallel is very strong in the case of functional programming languages which are also based upon the λ-calculus. The main difference is that in the case of a programming language, non-terminating functions are welcome.

To sum up

In type theory, a type itself can represent a proposition. Hence, an inhabitant of a type is also a proof. This is the *proposition-as-type* principle. Axioms of such type theory are given by the typing rules themselves.

Versatility of type systems

Nowadays, we observe a large diversity of type systems which for most of them extends Church's λ-calculus. While set theory proves to be effective to formalize mathematics on paper, mechanizing mathematics on a computer is a completely different task. One reason is that many details not necessary on paper prove to be essential to mechanize the proof on a computer. An example comes from statements which are trivial for humans, but are not if they are formalized in one of these systems. For example, a human can see trivially that $a + c + cd + da = (c + a)(1 + d)$, however for a computer this fact is not trivial since it involves several properties on operators $+$ and $\times$. Hence, to prove this fact formally, one needs to detail the computation:

(1)	$a + c + cd + da$	Main Hypothesis	
(2)	$a + (c + cd) + da$	Associativity of addition	1
(3)	$a + (c(1 + d)) + da$	Distributivity of multiplication over addition	2
(4)	$a + da + c(1 + d)$	Commutativity of addition	3
(5)	$a + ad + c(1 + d)$	Commutativity of multiplication	4
(6)	$a(1 + d) + c(1 + d)$	Distributivity of multiplication over addition	5
(7)	$(a + c)(1 + d)$	Distributivity of multiplication over addition	6

In some cases, deciding whether two mathematical expressions are propositionally equivalent is decidable. However, the logician Kurt Gödel proved that this was not possible in general [God92]. Another idea to avoid such painful details is to enrich the conversion $\equiv$ so that an equality as the one above becomes true by computation. Hence, if the user detects that two expressions are equal by computation, it can let the computer find it out. But this is not always possible, for example the commutativity is a property which is hard to turn into a computation directly.

Today, there is no consensus on the typing rules (the axioms of the theory) as well as the conversion relation (computational axioms) and this is why we observe such a diversity of type systems. A trade-off is made between having an expressive type system with a rich conversion and having a type system where the type checking is decidable (by an algorithm) and fast.

To sum up

During the last 50 years, many formal systems based on type theory and λ-calculus emerged to propose new foundations for mathematics.

Proof systems

Proof systems are software tools which allow humans to write mathematics on a computer. The main task of a proof system is to check that a proof written by a human is correct. Many proof systems today are based upon a type theory using the proposition-as-type principle. Hence, checking whether a proof is valid is the same as checking that the proof (as a program) has the expected type. As we saw in the previous section, formalizing a proof in a type system requires many cumbersome details and this is why these systems implement in general a higher-level language for the user to write proofs. Then, these proofs are *compiled* (or refined) to a judgment in the type system that can be checked by the system. Such higher-level language has generally two components:

- A vernacular that gives specific instructions to the proof system. In general the vernacular is used to structure the different proofs as a library so that proofs can be reused in other projects.
- A tactic language to write proofs without giving all the information needed. The missing pieces of information are reconstructed automatically by the system

We observe also a large diversity of high-level languages. Today, each proof system comes with its own high-level language. However, they generally all have a vernacular and a tactic language. Some of these type systems that will be mentioned in this document are Agda [Nor09], Coq [BGG+14], the Higher-Order Logic family [Har09, SN08, NPW02], Lean [dMKA+15], Matita [ASCTZ07] and PVS [ORS92].

To sum up

Formal systems have been implemented on a computer as *proof systems*. A proof system can be effectively used by a human to formally prove mathematical statements. However, such a proof is written in a high-level language where pieces of information omitted by the user are reconstructed by the proof system.

Achievements of formal proofs and formal verification:

Formalizing a proof in a well-known proof system provides one of the highest confidence we have today about the validity of a proof. During the last decades, proof systems have been used effectively to formalize very complex mathematical proofs:

- Hales theorem (formalized in HOL-Light) [Hal05]
- Four colors theorem (formalized in Coq) [Gon]
- Feit-Thompson theorem (formalized in Coq) using the math-component library [GAA+13]

The parallel of type systems with programming languages make proof systems based upon type theory also suitable to prove complex pieces of software. In this area, there are also great achievements:

- Sel4, a micro-kernel for an operating system (formalized in Isabelle/HOL) [KEH+09]
- The detect-and-avoid system for unmanned aircraft system developed by NASA (formalized in PVS) [ORSVH95]
- Compcert, a C compiler certified (formalized in Coq) [Ler16]
- CakeML, a certified compiler for a functional programming language (formalized in HOL4) [KMNO14]
- The correctness of the automatic Paris metro line 14 (formalized in B) [BBFM99]

These lists are not exhaustive. The reader may found a deeper inspection of the use of proof assistants in [Geu09].

The people and the time involved to formalize all of theses projects was huge. The order of magnitude is about several *person-years*. This is why such achievement is currently reserved for the research community. However, a lot of research is devoted to make proof systems easier to use.

To sum up

Proving a theorem on a proof system is difficult and takes a huge amount of time with respect to a proof on paper. Several achievements show that such systems are scalable.

Interoperability between proof systems

As for programming languages, each proof system comes with its own standard library. But this also means that the same theorem may be proved many times, once for each proof system. Because formalizing a theorem is difficult and may take several *person-years*, it is interesting to look for solutions where a theorem could be shared between proof systems once it has been proved once in one of them. However, sharing a proof from one proof system to another is a complex task. It raises theoretical problems:

- Proof systems do not use the same logic (the same type system for example)
- A theorem in one logic may not be provable in another, or even inconsistent with another proof system
- Vernacular and tactics are different from one proof system to another

But in addition, there are also practical issues:

- The number of translations is quadratic: A translation for every pair of proof system
- How to write these translations? In which programming language?
- How to maintain these translations?

Therefore, addressing the problem of interoperability requires to find both, theoretical and practical solutions.

To sum up

The counterpart of having many formal systems is that the same theorem is proved many times, once for for each *proof system*. Interoperability aims at sharing theorems between proof systems.

Logical Frameworks

Logical Frameworks are a particular kind of logical system (most of the time, type system) where it is possible to *embed* other logical systems in it. Actually, Predicate Logic from Frege is a logical framework because one may express other logical systems as theories. Several other logical frameworks appeared during the twentieth century.

To sum up

Logical Frameworks are systems into which other logical systems can be expressed.

Our interest behind logical frameworks is that they solve the quadratic number of translations issue we mentioned at the previous section: If every system can be embedded into one logical framework, the number of translations becomes linear. Such mechanism is already used in other applications:

- LLVM [LA04] which is a low-level language that makes interoperable high-level programming languages with different assembly languages
- Pandoc [Dom14] which makes text formats interoperable (LaTeX, Markdown, HTML, ...) by using a common internal language

To sum up

Logical Frameworks are good candidates to make proof systems interoperable.

One logical framework of particular interest is LF [HHP93a] which is a very simple type system with dependent types. Dependent types is a feature where a type may depend on the value of an object: This is the case for the type of matrices which are indexed by their size: $(2, 2)$-matrices. LF has been shown as an interesting logical framework from the theoretical point of view. However, the computational equality in LF is not very expressive and embeddings from other systems do not scale with real proofs.

A solution to overcome this issue is to enhance LF with an abstract computational equality. This new logical framework is called λΠ-CALCULUS MODULO THEORY. This way it becomes easier to embed other systems in λΠ-CALCULUS MODULO THEORY that scale on real proofs. The problem with using an arbitrary conversion is that checking that a proof is correct is not always decidable. However, if this abstract conversion can be decided by a set of rewrite rules which has good properties (termination and confluence) then this process becomes decidable.

To sum up

λΠ-CALCULUS MODULO THEORY is a logical framework with an abstract notion of computation which scales well on real proofs.

Content of this Thesis

In this thesis, we tackle the problem of interoperability between proof systems. In particular, we put our focus on the type systems underlying the proof systems and not the high-level languages. We have decided to use a logical framework as our *corner stone* to make proofs interoperable. As we will see, the choice of a logical framework is important since it will guide our translations but also the tools we use. Our choice was to use λΠ-CALCULUS MODULO THEORY as our logical framework for several reasons:

- This logical framework is very expressive and many systems can be embedded into it in a scalable way [CD07, Ass15b, Cau16a]
- It has an implementation called DEDUKTI [ABC+16] where the abstract conversion can be decided (using rewrite rules)
- We observe that in practice, an embedding into λΠ-CALCULUS MODULO THEORY with an abstract conversion can also be decided with rewrite rules
- Embeddings have been used to embed effectively many proofs coming from different systems such as: MATITA [Ass15b], HIGHER-ORDER LOGIC [AB15],FOCALIZE [Cau16a]
- The code behind DEDUKTI is short (about 3000 lines of OCaml code) and it makes it very easy to adapt to our own needs

We have split this manuscript into two parts. A first part entitled **Meta-theory of Cumulative Types Systems and their embeddings to the λΠ-CALCULUS MODULO THEORY**: it presents theoretical results about interoperability between proof systems. In particular we explain how Cumulative Type Systems (CTS) provide a good skeleton for interoperability between proof systems. A second part entitled **Interoperability in Dedukti: A case-study with Matita's arithmetic library**: it explains how we were able in practice to write a semi-automatic translation from MATITA to STT∀ (a constructive version of HIGHER-ORDER LOGIC) in DEDUKTI and then, to export these proofs to different systems: COQ, LEAN, MATITA, PVS and OPENTHEORY. To our knowledge, it is the first time that a library of proofs can be shared by 6 different proof systems (including DEDUKTI).

Our thesis

In this thesis, we explain how λΠ-CALCULUS MODULO THEORY provides a theoretical and practical solution to make proof systems interoperable.

In page 20, we present a general picture representing the content of this thesis. In this picture we have represented logics as ellipses. Here is the legend of this picture:

- CTS related type systems
- A CTS specification
  - $\mathscr{C}$ and $\mathscr{D}$ are any CTS specification
  - $\mathscr{S}$ is the CTS specification associated to STT∀
  - $\mathscr{M}$ is the CTS specification associated to MATITA
- λΠ-CALCULUS MODULO THEORY or DEDUKTI type systems
- Logics supporting common proof systems
- STT∀'s logic

$D[X]$ The logic X in DEDUKTI

$\mathscr{M} + I$ A subset of MATITA with the CTS specification of MATITA and inductive types

⟶ Encoding from one logic to another

⇢ Partial encoding (may not be sound)

$\mathrm{WS}_n(\Gamma \vdash_{\mathscr{C}} t : A)$ Judgments which have a well-structured derivation tree (Definition 3.1.2)

Meta-theory of Cumulative Types Systems and their embeddings to the λΠ-CALCULUS MODULO THEORY

Each type system has several features and its own degree of complexity. However, we observe that many of these type systems share a common part which is the λ-calculus even if the type systems may differ for this part. We have found that CTS (which extends Pure Type Systems with a subtyping relation) provide a good framework to study the differences between these type systems. We also discovered that the understanding of interoperability between CTS was the key to understand interoperability between proofs expressed in a type system using the proposition-as-type principle. Because we use the logical framework λΠ-CALCULUS MODULO THEORY, it is essential to understand the embedding of CTS into λΠ-CALCULUS MODULO THEORY. We also introduce one particular CTS called STT∀. STT∀ extends λ-HOL with prenex polymorphism (quantification over a type is allowed only at the head of a proposition or a type). Our interest for STT∀ lies in the fact that the type system behind STT∀ is a subset of many other type systems. Thus, it makes STT∀ a nice target for interoperability to export proofs to other systems.

The main results of this part are:

- A decidable procedure to decide whether a proof can be translated from one Cumulative Type System to another.
- A sound embedding of Cumulative Type Systems into the λΠ-CALCULUS MODULO THEORY logical framework.

Chapter 1: This chapter introduces Cumulative Type Systems (CTS), a logical framework which is behind many concrete proof systems today. The particularity of CTS as a logical framework, is that its type system is actually a family of type systems parameterized with a *specification*. CTS extends PTS with a cumulativity relation on sorts. In this chapter, we discuss why CTS provide an interesting framework to study the various proof systems available today since in general, the theories behind these systems can be reformulated as extensions of some CTS.

Chapter 2 In this chapter, we investigate *interoperability* between CTS. In particular, we extend the usual notion of interoperability (given as a sort-morphism) with new definitions for equivalences between CTS specifications. These new definitions allow us to conclude that any CTS is equivalent to a functional and injective CTS.

In a second part of this chapter, we investigate an incomplete procedure to decide whether a judgment from one CTS can be embedded into another. This method relies on the generation of a free CTS. The incompleteness comes from the fact that some pieces of information are missing in a CTS judgment that needs to be reconstructed.

Chapter 3: This chapter introduces notion of well-structured derivation trees. The idea behind this predicate is to attach a level to a derivation tree of a CTS which gives an induction principle compatible with subject reduction. We show that this induction principle gives a simple proof to solve difficult problems such as expansion postponement and the equivalence between an implicit conversion and an explicit (or typed) conversion. While we were not able to prove that any CTS derivation tree is well-structured, we have empirically verified that the derivation trees we manipulated in the second part of this thesis are well-structured. We also investigate this conjecture and give some insights behind the difficulty of this conjecture.

Chapter 4: This chapter introduces bi-directional CTS. In a bi-directional CTS, the typing judgment is split in two: an inference judgment without cumulativity, and a checking judgment with cumulativity. In this system, the cumulativity can only be used during an application or at the end of a proof. We prove that for a large class of CTS called *CTS in normal form*, any well-structured proof can be translated into a bi-directional CTS proof. Bi-directional CTS are used in Chapter 6 to express the translation of CTS into λΠ-CALCULUS MODULO THEORY.

Chapter 5: This chapter introduces PTS modulo which enrich PTS with an abstract conversion generated by equations. This system is a reformulation of the systems presented by Frédéric Blanqui in [Bla01] where the judgmental equality is enriched incrementally. In particular, we are interested in λΠ-CALCULUS MODULO THEORY which is the PTS modulo that corresponds to LF, a minimalist type theory with dependent types. Cousineau & Dowek showed in [CD07] that any PTS could be embedded in λΠ-CALCULUS MODULO THEORY and as such, any PTS modulo. Hence, λΠ-CALCULUS MODULO THEORY is a logical framework which generalizes PTS. However, type checking in λΠ-CALCULUS MODULO THEORY is not decidable.

Chapter 6: In this chapter, we investigate the embedding of CTS into λΠ-CALCULUS MODULO THEORY. We define a translation function for all CTS in normal form which generalizes Ali Assaf's results [Ass15b]. In particular, we use a *cast operator* which generalizes the explicit *lift operator* of Ali Assaf. This solves a conservativity issue we had identified in Ali Assaf's embedding and we conjecture that this new embedding is conservative. We prove the soundness

of this encoding for any well-structured derivation tree as well as a *shape-preserveness* property which proves that your encoding is not trivial.

Chapter 7: This chapter introduces STT∀, a constructive version of Simple Type Theory with prenex polymorphism and type constructors. STT∀ will be used in the second part of this thesis as the target logic to translate arithmetic proofs coming from Matita as discussed in Chapter 12. The benefit of STT∀ is that proofs in this logic can be easily exported in many other proof systems. Besides, we show that STT∀ is also a CTS. Seeing STT∀ as a CTS allows to derive its embedding into λΠ-calculus modulo theory for free using results from Chapter 6.

Interoperability in Dedukti: A case-study with Matita's arithmetic library

The second part of this thesis is entitled **Interoperability in Dedukti: A case-study with Matita's arithmetic library**; it explains the tools we used to translate, effectively and in a semi-automatic way arithmetic proofs originally written in the Matita proof system to four other proof systems which are: Coq, Lean, PVS and OpenTheory. Our translation is actually cut into smaller parts to deal with different features of the Matita's type system. In the first part, we have seen that the CTS framework was the main component, but because the type system of Matita implements the Calculus of Inductive Constructions logic, we also had to cope with inductive types and recursive functions.

While all of these translations could be written in a programming language such as OCaml, we have decided to use a meta language for Dedukti called Dkmeta. We have made this choice because we realized that proof translations were hard to understand, verbose and really hard to maintain in a programming language. Indeed, every time there was a new version of Dedukti, or every time there was a change in one of the encoding used, the translations were broken. Proof translations in Dkmeta are more robust than in OCaml and easier to fix.

The main result of this part is:

- A semi-automatic procedure to translate proofs from one system to another. It was applied on Fermat's little theorem written in Matita and translated to four other different formal systems: Coq, Lean, OpenTheory and PVS.

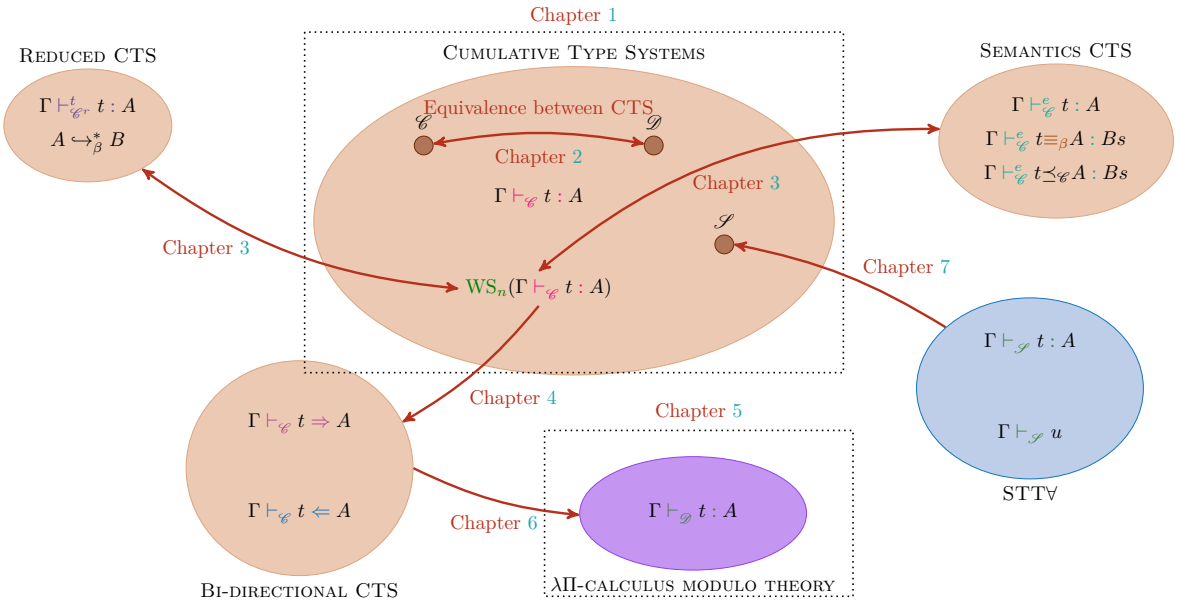
Chapter 8: This chapter presents Dedukti, an implementation of λΠ-calculus modulo theory where the type checking is decidable. The equations are provided as rewrite rules. We introduce the syntax of Dedukti that will be used for the remaining part of this thesis. We show how the embeddings we defined in Chapter 6 and Chapter 7 can be formulated in Dedukti via rewrite rules.

Chapter 9: This chapter introduces a tool we have developed for Dedukti. This tool uses the rewrite rules of Dedukti as a meta language to manipulate proofs. Besides rewriting, this language features a quote/unquote mechanism to enrich the expressivity of the language. The quoting mechanism allows to overcome limitations of the rewrite engine of Dedukti but also helps defining meta programs which rely on types. We argue, using several examples, that having a meta language such as the one provided by Dkmeta is a powerful approach to write many proof transformations in a simple way.

Chapter 10: This chapter introduces another tool for Dedukti. Universo implements the incomplete procedure presented in Chapter 2 to decide whether a judgment from one CTS can be embedded into another. Because Universo relies on an SMT solver (currently Z3) it is not clear whether this tool can scale with big libraries of proofs. However, we have successfully applied Universo on the arithmetic library of Matita.

Chapter 11: This chapter presents a translation via Dedukti of the proof of Fermat's little theorem written in Matita (an implementation of the Calculus of Inductive Constructions) to STT∀. We claim that this translation can be fully automatized even if at the time of writing it is not completely done. This chapter introduces other tools developed for Dedukti: Dkprune, which computes the set of minimal dependencies for a theorem and Dkpsuler which allows the instantiation of definitions.

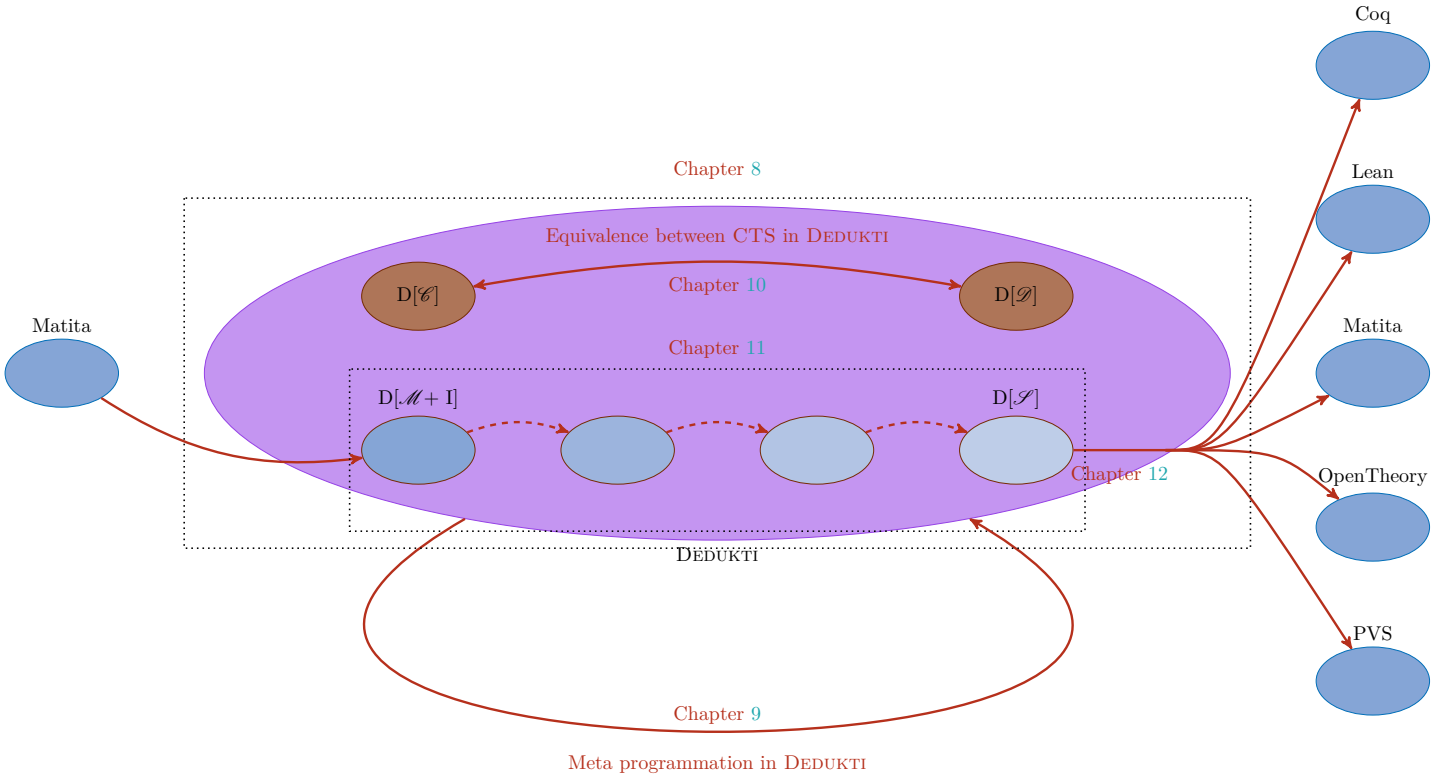
Chapter 12: This chapter explains how proofs written in STT∀ can be exported to different concrete systems. In our case, we have successfully exported Fermat's little theorem to Coq, Lean, Matita, PVS and OpenTheory [Hur11]. We have built a website around these exportations called Logipedia(`www.logipedia.science`), which provides a nice user interface to make these translations available. This project could be the start of an encyclopedia that shares formal proofs between various proof systems.



Meta-theory of Cumulative Types Systems and their embeddings to the λΠ-calculus modulo theory

---

Interoperability in Dedukti: A case-study with Matita's arithmetic library



Miscellaneous remarks to read this manuscript

In the first part of this manuscript, you may encounter proofs which are written this way:

◊ $\mathscr{C}^{\Rightarrow}_{var}$: $t = x$

	(1)	$\Gamma \vdash_{\mathscr{C}} x \Rightarrow A$	Main hypothesis	
	(2)	$\Gamma \vdash_{\mathscr{C}\Rightarrow}$ wf	Inversion on $\mathscr{C}^{\Rightarrow}_{var}$	1
	(3)	$x : A \in \Gamma$		
	(4)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}}$ wf	Induction hypothesis	2
	(5)	$x : [\![A]\!]_\Gamma \in [\![\Gamma]\!]$	Lemma 6.2.8	3
	(6)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} x : [\![A]\!]_\Gamma$	$\mathscr{R}_{var}$	4,5
	(7)	$x = [x]_\Gamma$	Definition of $[\cdot]$.	
	(8)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [x]_\Gamma : [\![A]\!]_\Gamma$	Congruence of equality	6,7
★	(9)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [t]_\Gamma : [\![A]\!]_\Gamma$	Definition of t	8

This proof should be read as follow:

- ◊ means we are doing a proof by case analysis or by induction.
- The first parameter (as $\mathscr{C}^{\Rightarrow}_{var}$) is the last rule of the derivation (if the case applies)
- The second parameter contains equalities specific to the case: $t = x$ in the example above. This means that the general statement of the theorem uses t and for the case $\mathscr{C}^{\Rightarrow}_{var}$, we know that t is the variable x.
- Each line of the proof is split into four columns: A number as an identifier of the line (local to the proof), a proposition, a justification for this proposition, identifiers used to apply the justification. For example to apply the derivation rule $\mathscr{R}_{var}$ at line 6 we need to provide two hypothesis. The first one is given by (4) and the second one by (5). If tooltips are activated (see below), you may click on the name of the rule to see the derivation rule and check that the lines referenced correspond to the premises.
- A line separate propositions which are obtained by different justifications
- A star indicates one objective to prove. Most of the time only the last line of the proof contains a star but sometimes a several propositions need to be proved.
- When a proposition is obtained indirectly, or because some details are missing, we may separate the proposition with two horizontal lines

The manuscript is readable on paper, but if you read this file as a PDF with Evince (or adobe), you may use tooltips to help your reading: Click me $\mathscr{C}^{\Rightarrow}_{var}$. If it works, you should see a tooltip as the one below:

$$\frac{\Gamma \vdash_{\mathscr{C}\Rightarrow} \text{wf} \qquad (x : A) \in \Gamma}{\Gamma \vdash_{\mathscr{C}} x \Rightarrow A}\ \mathscr{C}^{\Rightarrow}_{var}$$

Because some PDF viewers do not handle well tooltips, it may be better to read the manuscript without tooltips. To know whether tooltips are activated, look at the following message: tooltips unavailable ✗.

In the proof above, all the tooltips are not available for technical limitations and are available only for the real proof. In general a link which is not a reference number printed in blue has a tooltip.

Part I

Meta-theory of Cumulative Types Systems and their embeddings to the $\lambda\Pi$-calculus modulo theory

Chapter 1

Cumulative Type Systems

Cumulative Type Systems (CTS) were originally introduced by Bruno Barras in his PhD Thesis [Bar99a]. CTS extend Pure Type Systems [AGM92] with subtyping on sorts and co-domains for dependent products. CTS are a family of type systems which is parameterized with a specification. A lot of concrete systems (Coq [BGG+14], Matita [ASCTZ07], Agda [Nor09], the HOL family [Har09] [SN08] [NPW02], ...) can be seen as extensions of CTS. Hence CTS provide a common framework to study the properties for all of these systems and therefore a good basis to make these systems interoperable. The interest for CTS does not only hold for proof systems, but it can also be used for the theory behind functional programming languages, for example Haskell [Tho11], OCaml [LDF+18] or Idris [BRA13].

This chapter starts with introducing basic elements of type theory. First, we define the syntax of the lambda-calculus that we will use for all the systems involved in this thesis. Then we recall some of the computational rules which are implemented in concrete systems. Namely, α, β, η, δ and ζ relations. The computational rule ι which comes with inductive types will be introduced in Chapter 8.

Next, we introduce CTS specifications and define some interesting class of CTS specifications. Then, we finally introduce the type system of CTS. Before going on to the meta-properties of CTS we give first some examples of specifications which, for most of them, capture features that existed long before the definition of CTS. We explain in particular three features commonly found in most CTS specifications implemented today: Dependent types, polymorphism and higher-order types (or type constructors). We also give some examples of systems which are non-terminating. One in particular, $\lambda\star$ will be relevant for Chapter 2.

Finally, we introduce some classical meta-theoretical properties of CTS. The main one being subject reduction (or type preservation). We also make two remarks. The first one is about the type checking of CTS in general. This will be one of our motivations for bi-directional CTS introduced in Chapter 4. The second remark is about the definition of subtyping. In particular, we show an equivalence with another definition for subtyping which removes the transitivity rule. This definition will be used in the equivalence proof of Chapter 4 and the soundness proof of Chapter 6.

1.1 Syntax

We introduce the syntax of CTS. The syntax is parameterized with a set of sorts $\mathcal{S}$. For this chapter, this set will be given by a CTS specification.

Variables	x	$\in \mathcal{V}$		
Sorts	s	$\in \mathcal{S}$		
Terms	t, u, M, N, A, B	$\in \mathcal{T}$	$::=$	$x \mid s \mid M\ N \mid \lambda x : A.\, M \mid (x : A) \to B$
Typing contexts	Γ	$\in \mathcal{G}$	$::=$	$\varnothing \mid \Gamma, x : A$

Figure 1.1: PTS syntax

Definition 1.1.1 (Syntax of terms)
The syntax of terms is defined in Fig. 1.1. It is parameterized with a set $\mathcal{S}$ and a set $\mathcal{V}$. We make the usual assumption that $\mathcal{V}$ is an infinite set with a decidable equality.

- *x is called a* variable*,*
- *s is called a* sort *(or* universe*),*
- *$M\ N$ is called an* application*,*
- *$\lambda x : A.\, M$ is called an* abstraction*,*
- *$(x : A) \to B$ is called a* product*,*

Remark 1 *In the literature, products are also written $(x : A)B$, $\forall x : A,\ B$ or $\Pi(x : A).\ B$.*

Remark 2 *Application is right-associative and product is left-associative. Hence, $(f\ a)\ b$ is written $f\ a\ b$ and $(x : A) \to ((y : B) \to C)$ is written $(x : A) \to (y : B) \to C$.*

Notation 1 *We overload the notation $\Gamma, x : A$ to also denote Γ, Γ' the concatenation of Γ and Γ'.*

$\lambda x : A.\, M$ and $(x : A) \to B$ are binders, which means that x is *bound* in M and in B. Expressing that a variable is bound is in general not easy to formalize (depending on the meta-language used). Below, we introduce usual definitions to express this notion formally. In particular, we need to refine the syntactic equality on terms to take into account that bound variables can be renamed. This intuition will be detailed in Paragraph 1.2.2. To cope with this issue, proof systems use most of the time the so-called De Bruijn indices [DB72], or Higher-Order Abstract Syntax (HOAS) [LR18].

Definition 1.1.2 (Free variables)
The set of free variables function $\mathrm{FV}(\cdot) : \mathcal{T} \to 2^{\mathcal{V}}$ is defined as usual [AGM92].

Example 1.1 *In $x\ (\lambda x : A.\, x\ y)$, the first occurrence of x (from left to right) is free as well as y. Hence, its set of free variables is $\{x, y\}$.*

Notation 2 *We define the notation $A \to B$ as $(x : A) \to B$, where $x \notin \mathrm{FV}(B)$.*

Definition 1.1.3 (Syntactic context)
A syntactic context *denotes a term with a hole. More formally a syntactic context can be describe by the following grammar*

$$\text{Syntactic contexts} \quad \dot{C} \quad ::= \quad [\cdot] \mid \lambda x : \dot{C}.\, M \mid \lambda x : A.\, \dot{C} \mid (x : \dot{C}) \to B \mid (x : A) \to \dot{C} \mid \dot{C}\ N \mid M\ \dot{C}$$

Figure 1.2: PTS syntax

$$\frac{t R t'}{t \hookrightarrow_R t'}\ \cdot$$

$$\frac{A \hookrightarrow_R A'}{\lambda x : A.\, t \hookrightarrow_R \lambda x : A'.\, t}\ \cdot^{l}_{\lambda} \qquad \frac{t \hookrightarrow_R t'}{\lambda x : A.\, t \hookrightarrow_R \lambda x : A.\, t'}\ \cdot^{r}_{\lambda}$$

$$\frac{t \hookrightarrow_R t'}{t\ u \hookrightarrow_R t'\ u}\ \cdot^{l}_{app} \qquad \frac{u \hookrightarrow_R u'}{t\ u \hookrightarrow_R t\ u'}\ \cdot^{r}_{app}$$

$$\frac{A \hookrightarrow_R A'}{(x : A) \to B \hookrightarrow_R (x : A') \to B}\ \cdot^{l}_{\Pi} \qquad \frac{B \hookrightarrow_R B'}{(x : A) \to B \hookrightarrow_R (x : A) \to B'}\ \cdot^{r}_{\Pi}$$

Figure 1.3: Contextual rule for an abstract relation R

1.2 Rewriting

Terms can be equipped with a notion of computation. Computation in type theory is expressed with a relation that we call rewriting relation. The main one being the β rewriting relation. It is well-known that this relation gives rise to a computational model which is Turing-complete [Ros39] for pure lambda terms (without types). However, β is not the only computational rule implemented in proof systems and many others exist. We survey here all the computational rules that we will mention in this thesis except one: ι which comes with inductive types which will be detailed in Section 8.4.

1.2.1 Rewriting relation

Definition 1.2.1 (Rewriting relation)
A rewriting relation is a relation over the set of terms $\mathcal{T}$.

Definition 1.2.2 (Rewriting relation stable by context)
A rewriting relation R is stable by syntactic context if it is stable by the rules given in Figure 1.3. Such a relation is generally denoted by $\hookrightarrow_R$. A redex *for a rewriting relation $\hookrightarrow_R$ is a term t such that there exists u with tRu.*

Remark 3 *If $t \hookrightarrow t'$, it is said that t* reduces *(or* computes*) to t'.*

Notation 3 *Given a rewriting relation $\hookrightarrow$, we denote:*

- *its inverse relation by* $\hookleftarrow$
- *its transitive and reflexive closure by* $\hookrightarrow^*$

Remark 4 *A left to right rewriting (as in* $t \hookrightarrow t'$*) is called a* reduction *(*t *reduces to* t'*) while a right to left rewriting (as in* $t \hookleftarrow t'$*) is called an* expansion *(*t *expands to* t'*).*

Definition 1.2.3 (Congruence generated by a rewriting relation)
The congruence *generated by a relation* R *is the smallest relation which includes* R *and is stable by transitivity, symmetry, reflexivity and syntactic context. For a rewriting relation* $\hookrightarrow_R$*, the congruence is denoted* $\equiv_R$*.*

We redefine below some common properties related to rewriting relations.

Definition 1.2.4 (CR)

A rewriting relation $\hookrightarrow$ *is said* Church-Rosser *(CR) (or* confluent*) if* $u \hookleftarrow^* t \hookrightarrow^* v$ *then there exists* w *such that* $u \hookrightarrow^* w \hookleftarrow^* v$*.*

Given a term t *such that* $u \hookleftarrow^* t \hookrightarrow^* v$*,* (u, v) *is called a* critical pair[1]*. A critical pair* (u, v) *is said* joinable *if there exists* w *such that* $u \hookrightarrow^* w \hookleftarrow^* v$*.*

Remark 5 *Confluence can be reformulated as: All the critical pairs are joinable.*

Definition 1.2.5 (NF)
A term t *is in normal form (NF) with respect to a rewriting relation* $\hookrightarrow$ *if there is no* t' *such that* $t \hookrightarrow t'$*.*

Definition 1.2.6 (WN)
A rewriting relation $\hookrightarrow$ *is said* weakly normalizing *(WN) if for all terms* t*, there exists* u *such that* $t \hookrightarrow^* u$ *and NF(*u*).*

Definition 1.2.7 (SN)
A rewriting relation $\hookrightarrow$ *is said* strongly normalizing *(SN) if for all terms* t*, there is no infinite sequence* $(t_n)_{n\in\mathbb{N}}$ *such that* $t_0 = t$ *and for all* n*,* $t_n \hookrightarrow t_{n+1}$*.*

Remark 6 *As explained in [Len06], this definition tends to be classical. Indeed, if one wants to prove that a rewriting relation is not SN, we would not obtain directly as a witness an infinite sequence of* t_n*. Instead we get the double negation that this sequence does exist. To solve this issue, there is a way to capture that every term is SN with an inductive definition which roughly defines the property of being SN first on the normal forms and then, if all the reducts of a term* t *are SN, then* t *itself is SN. In this thesis, such difference is not that important since we won't study the normalization of CTS.*

[1] This definition is slightly different from the one found in the litterature as in [BN99]. For this manuscript this simpler definition is enough.

1.2.2 α relation

In general, in mathematics, we consider the two following functions $x \mapsto x$ and $y \mapsto y$ as equal, the name x or y being irrelevant. However, their equivalent in the syntax presented in Fig 1.1 represent these two functions as two different objects: $\lambda x.\, x$ and $\lambda y.\, y$. The common way to solve this problem[2] is to define an equivalence relation and to reason modulo this equivalence relation. Such equivalence is generally called α. Actually, defining this relation is not trivial and has been done many times. Below, we introduce only our notations, all the definitions can be found in [AGM92].

Notation 4 (Substitution) *The function $\cdot\{\cdot \leftarrow \cdot\} : \mathcal{T} \to \mathcal{V} \to \mathcal{T} \to \mathcal{T}$ denotes the substitution function on terms. This definition can be naturally extended to typing contexts.*

Remark 7 *Substitution needs that $\mathcal{V}$ to be infinite to be well-behaved because fresh names need to be generated.*

Example 1.2 *We give some examples of a substitution applied to some terms:*

- $x\{x \leftarrow z\} = z$
- $x\{y \leftarrow z\} = x$ *(if $x \neq y$)*
- $(\lambda x : A.\, x)\{x \leftarrow z\} = \lambda x : A\{x \leftarrow z\}.\, x$
- $((x : A) \to y)\{y \leftarrow z\} = (x : A\{y \leftarrow z\}) \to z$

Definition 1.2.8 (modulo-α)
We denote $\equiv_\alpha \subseteq \mathcal{T} \times \mathcal{T}$ the α relation.

Theorem 1.2.1 *The relation $\equiv_\alpha$ is a congruence.*

Example 1.3 *Some examples of terms (un)-equals modulo α:*

- $x \not\equiv_\alpha y$ *(if $x \neq y$)*
- $\lambda x : A.\, x \equiv_\alpha \lambda y : A.\, y$
- $\lambda x : A.\, (\lambda x : B.\, x) \equiv_\alpha \lambda z : A.\, (\lambda y : B.\, y)$

In the remainder of this thesis, we will always compare terms modulo this equivalence relation and write $=$ instead of $\equiv_\alpha$.

1.2.3 β relation

β computes the result of a function applied to an argument as when the function $x \mapsto x^2$ is applied to the natural number 2. This function computes (or reduces) to 4. In our syntax this would be written $\left(\lambda x.\, x^2\right)\ 2 \hookrightarrow_\beta 4$.

Definition 1.2.9 ($\hookrightarrow_\beta$)
The relation $\hookrightarrow_\beta$ is defined as the congruence generated by $((\lambda x : A.\, M)\ N)\, \beta\, M\{x \leftarrow N\}$.

[2] if the meta-theory does not have a primitive notion of binders

In general, a term may contain several β redexes. Confluence is an important property which expresses that the order in which we apply the reductions does not really matter because given two sequences of reductions, there is always a way to finish the sequences to get the same result at the end.

Theorem 1.2.2 (Confluence of β) *β is CR: If $u \hookleftarrow_\beta^* t \hookrightarrow_\beta^* v$ then there exists w such that $u \hookrightarrow_\beta^* w \hookleftarrow_\beta^* v$.*

1.2.4 η relation

Definition 1.2.10 ($\hookrightarrow_\eta$)
The relation $\hookrightarrow_\eta$ is defined as the congruence generated by $(\lambda x : A.\, M\ x)\ \eta\ M$ where $x \notin \mathrm{FV}(M)$.

Definition 1.2.11 ($\hookrightarrow_{\beta\eta}$)
The relation $\hookrightarrow_{\beta\eta}$ is defined as the union of $\hookrightarrow_\beta$ and $\hookrightarrow_\eta$.

Theorem 1.2.3 ($\hookrightarrow_{\beta\eta}$ is not CR) *[Geu93] The relation $\hookrightarrow_{\beta\eta}$ is not Church-Rosser.*

Proof *This counterexample is due to Nederpelt. We have the following critical pair:*

$$\lambda x : \mathbb{N}.\, x \hookleftarrow_\beta \lambda x : \mathbb{N}.\, (\lambda y : \mathbb{N} \to \mathbb{N}.\, y)\ x \hookrightarrow_\eta \lambda y : \mathbb{N} \to \mathbb{N}.\, y$$

It is not joinable since $\mathbb{N}$ is not convertible to $\mathbb{N} \to \mathbb{N}$.

Hence, to recover the Church-Rosser property, one needs to take into account the fact the terms we are considering are well-typed [Geu93]. In Section 1.4, we define the *typing system* as a ternary relation which defines a notion of *well-typed terms*. Geuvers proved that for well-typed terms, the CR property holds for $\hookrightarrow_{\beta\eta}$:

Theorem 1.2.4 (Confluence of $\beta\eta$ for well-typed term) *[Geu93] If $\Gamma \vdash_{\mathscr{C}} t : A$ and $u \hookleftarrow_{\beta\eta}^* t \hookrightarrow_{\beta\eta}^* v$ then there exists w such that $u \hookrightarrow_{\beta\eta}^* w$ and $v \hookrightarrow_{\beta\eta}^* w$.*

Moreover, we will see in Section 1.7, that the behavior of η with subtyping can be quite surprising.

1.2.5 δ relation

In practice, it is very useful to have the ability to give a name to a term, and as such, having a mechanism for definitions. The classical way to handle such mechanism is to enhance the typing context with a new construction for definitions.

Definition 1.2.12 (Typing context extension with global definitions)
We consider the syntax extension of CTS presented in Figure 1.4. The meaning of $f : A = t$ is that f is the name for the term t of type A. We prefer to have another set of names for definitions $\mathcal{F}$ to avoid any ambiguity. We assume also that $\mathcal{F}$ is infinite.

Such mechanism leads to a rewriting relation (which depends on the typing context).

Definition 1.2.13 (δ relation)
Given a typing context Γ in the extended syntax, the δ rewriting relation ($\hookrightarrow_{\delta_\Gamma}$) is defined as the smallest relation that includes

$$f \hookrightarrow_{\delta_\Gamma} t$$

if there exists A such that $f : A = t \in \Gamma$ and stable by typing context. In general, we omit the typing context Γ in the notation and simply write $\hookrightarrow_\delta$.

Constants	f	$\in \mathcal{F}$		
Terms	t, u, M, N, A, B	$\in \mathcal{T}$	$::=$	$\cdots \mid f$
Typing contexts	Γ	$\in \mathcal{G}$	$::=$	$\varnothing \mid \Gamma, x : A \mid \Gamma, f : A = t$

Figure 1.4: Syntax extension of CTS with definitions

Terms	t, u, M, N, A, B	$\in \mathcal{T}$	$::=$	$\cdots \mid \mathbf{let}\,(x\colon A) := t\,\mathbf{in}\,u$

Figure 1.5: Syntax extension of CTS with local definitions

Definition 1.2.14 (Rewriting modulo β and δ)
The relation $\hookrightarrow_{\beta\delta}$ is defined as the union of $\hookrightarrow_\beta$ and $\hookrightarrow_\delta$.

Theorem 1.2.5 ($\hookrightarrow_{\beta\delta}$ is CR) *[Bar99a] The rewrite relation $\hookrightarrow_{\beta\delta}$ is CR.*

1.2.6 ζ relation

Having global definitions via the mechanism of δ rewriting is not enough in practice: while proving a lemma, we may introduce intermediate assertions, in other words, a cut [Len06]. With dependent types, a local cut cannot always be translated as a β redex. To give the ability to introduce a local assertion we extend the language with a local definition mechanism.

Definition 1.2.15 (Typing context extension with local definitions)
We consider the syntax extension of CTS with local definitions presented in Figure 1.5. In $\mathbf{let}\,(x\colon A) := t\,\mathbf{in}\,u$, the variable x is bound in u. Hence, one needs also to extend $\equiv_\alpha$ to take into account this new binder.

Local definitions lead to the ζ rewriting relation:

Definition 1.2.16 (ζ relation)
The ζ rewriting relation is defined as the smallest relation which includes

$$\mathbf{let}\,(x\colon A) := t\,\mathbf{in}\,u \hookrightarrow_\zeta u\,\{x \leftarrow t\}$$

and stable by typing context.

In practice, proof systems use a combination of these rewriting relations which enhance the expressivity of the computation. For example, the Coq system [BGG+14] or the Matita system [ASCTZ07] uses the $\beta\eta\delta\zeta$ rewriting relations and is modulo α.

In the following, the type system is defined only with β and α. For the other relations, we will not detail the extensions here. However, we will mention them in the second part of this manuscript since we will deal with concrete systems such as Matita.

1.3 Cumulative Type Systems specification

The typing system of CTS is parameterized with a specification. Hence CTS are actually a family of type systems and as such constitute a logical framework.

Definition 1.3.1 (CTS specification)
A CTS specification *is a quadruple* $\mathscr{C} = (\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{C})$ *where:*

- $\mathcal{S}$ *is a set of constants called* sorts*,*
- $\mathcal{A} \subseteq \mathcal{S} \times \mathcal{S}$ *is a relation called* axioms*,*
- $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$ *is a relation called* rules*.*
- $\mathcal{C} \subseteq \mathcal{S} \times \mathcal{S}$ *is a relation called* cumulativity*.*

Notation 5 *Given a CTS specification* $\mathscr{C} = (\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{C})$*, we use the notation* $\mathcal{S}_{\mathscr{C}}$ *(resp.* $\mathcal{A}_{\mathscr{C}}$,$\mathcal{R}_{\mathscr{C}}$ *and* $\mathcal{C}_{\mathscr{C}}$*) to refer to the set* $\mathcal{S}$ *(resp.* $\mathcal{A}$*,* $\mathcal{R}$ *and* $\mathcal{C}$*) of the specification* $\mathscr{C}$*.*

Notation 6 $\mathcal{C}^*_{\mathscr{C}}$ *is the reflexive and transitive closure of* $\mathcal{C}_{\mathscr{C}}$*.*

Definition 1.3.2 (Top-sort ($\mathcal{S}^{\top}_{\mathscr{C}}$))
A sort s *is called a* top-sort *if there is no* s' *such that* $(s, s') \in \mathcal{A}$*. The set of top-sorts is written* $\mathcal{S}^{\top}_{\mathscr{C}}$*.*

From now on, given a specification $\mathscr{C}$, we identify the set of sort $\mathcal{S}$ of the syntax presented in Fig. 1.1 with the set $\mathcal{S}_{\mathscr{C}}$, hence the syntax also depends on the specification.

Definition 1.3.3 (PTS specification)
A PTS is a particular case of CTS where the cumulativity relation $\mathcal{C}$ *is empty.*

Definition 1.3.4 (Underlying PTS)
For every CTS $\mathscr{C}$*, there is an underlying PTS* $\mathscr{P}$ *which is defined as the same specification as* $\mathscr{C}$ *except that* $\mathcal{C}_{\mathscr{P}} = \emptyset$*.*

Notation 7 *We use the letter* $\mathscr{P}$ *to refer to a PTS specification and* $\mathscr{C}$ *to a CTS specification.*

Definition 1.3.5 (Finite CTS specification)
A CTS specification $\mathscr{C}$ *is said* finite *if* $\mathcal{S}_{\mathscr{C}}$ *is.*

Definition 1.3.6 (Functional CTS)
A CTS specification is said functional *if the relations* $\mathcal{A}$ *and* $\mathcal{R}$ *are functional.*

Definition 1.3.7 (Injective CTS)
A CTS specification $\mathscr{C}$ *is said* injective *if:*

- *For all* s_a, s_b, s_c*,* $(s_a, s_b) \in \mathcal{A}_{\mathscr{C}} \wedge (s_c, s_b) \in \mathcal{A}_{\mathscr{C}} \Rightarrow s_a = s_c$
- *For all* s, s_a, s_b, s_b*,* $(s, s_a, s_b) \in \mathcal{R}_{\mathscr{C}} \wedge (s, s_c, s_b) \in \mathcal{R}_{\mathscr{C}} \Rightarrow s_a = s_c$

Definition 1.3.8 (Semi-Full CTS)
A CTS specification $\mathscr{C}$ *is said* semi-full *if for all* s_1*, if there exists* s_2, s_3 *such that* $(s_1, s_2, s_3) \in \mathcal{R}_{\mathscr{C}}$ *then for all* s_2*, there exists* s_3 *such that* $(s_1, s_2, s_3) \in \mathcal{R}_{\mathscr{C}}$*.*

Definition 1.3.9 (Full CTS)
A CTS specification $\mathscr{C}$ *is said* full *if for all* s_1, s_2*, there exists* s_3 *such that* $(s_1, s_2, s_3) \in \mathcal{R}_{\mathscr{C}}$*.*

$$\frac{(s,s') \in \mathcal{A}}{s \lhd_{\mathcal{S}_{SC}} s'} \qquad \frac{s \lhd_{\mathcal{S}_{SC}} s' \qquad s' \lhd_{\mathcal{S}_{SC}} s''}{s \lhd_{\mathcal{S}_{SC}} s''} \qquad \frac{s \lhd_{\mathcal{S}_{SC}} s' \qquad (s',s'') \in \mathcal{C}}{s \lhd_{\mathcal{S}_{SC}} s''}$$

Figure 1.6: Ordered relation

Predicativity and impredicativity: Many specifications behind concrete systems (such as The Calculus of Inductive Constructions) identify a particular sort which is inhabited by propositions. For several reasons that we do not detail here[3], these systems make this sort impredicative. Informally, a sort is impredicative if we can build objects in this sort by quantifying over a *larger* sort. However, the meaning of what is a larger sort is not really clear in general. To define this notion properly, we define first a notion of ordered specification[4].

Definition 1.3.10 (Ordered specification)
We define $\lhd_{\mathcal{S}_{SC}}$ the smallest relation defined by the rules in Figure 1.6 (as in [Las12]). If this smallest relation is a strict order, we say that the CTS specification is ordered.

Definition 1.3.11 (Impredicative sort)
If a specification $\mathscr{C}$ is ordered, then a sort s is said impredicative *if there exists s', s'' such that $(s', s'', s) \in \mathcal{R}_{\mathscr{C}}$ where $s \lhd_{\mathcal{S}_{SC}} s'$ or $s \lhd_{\mathcal{S}_{SC}} s''$.*

Definition 1.3.12 (Predicative CTS)
A CTS is said predicative *if there is no impredicative sort.*

It is important to mention though that in opposition to the definition of predicativity given in [Las12], we allow an ordered specification CTS which is not well-founded. The lack of theorems about predicativity does not help us to decide whether the ordered specification should be well-founded.

Decidable specifications While meta-theory of CTS can be formulated for any specification, in practice–especially to have decision procedures–we will restrict ourselves to decidable specifications.

Definition 1.3.13 (decidable CTS specification)
A CTS specification is decidable if:

- *the equality on $\mathcal{S}$ is* decidable
- *$\mathcal{A}, \mathcal{R}, \mathcal{C}$ are decidable relations (membership is decidable)*
- *Given s, knowing if there exists s' such that $(s, s') \in \mathcal{A}_{\mathscr{C}}$ is decidable*
- *Given s_1, s_2, knowing if there exists s such that $(s_1, s_2, s) \in \mathcal{R}_{\mathscr{C}}$ is decidable*

[3] An interesting discussion about impredicativity: http://lists.seas.upenn.edu/pipermail/types-list/2019/002150.html

[4] Our definition is more general than *dependence relation* introduced in [Las12]

$$\frac{A \equiv_\beta B}{A \preceq_{\mathscr{C}} B}\ \preceq_{\equiv_\beta} \qquad \frac{(s, s') \in \mathcal{C}^*_{\mathscr{C}}}{s \preceq_{\mathscr{C}} s'}\ \preceq_{\mathcal{C}^*_{\mathscr{C}}} \qquad \frac{B \preceq_{\mathscr{C}} B'}{(x:A) \to B \preceq_{\mathscr{C}} (x:A) \to B'}\ \preceq_{\Pi}$$

$$\frac{A \preceq_{\mathscr{C}} B \qquad B \preceq_{\mathscr{C}} C}{A \preceq_{\mathscr{C}} C}\ \preceq_{trans}$$

Figure 1.7: CTS subtyping relation

1.4 Typing

The typing relation of CTS is defined in two steps. First, we introduce the subtyping notion of CTS as a judgment $A \preceq_{\mathscr{C}} B$ which extends β conversion with a subtyping relation generated by $\mathcal{C}$. Second, we define the typing system using two judgments: $\Gamma \vdash_{\mathscr{C}} t : A$ meaning that t is of type A in the typing context Γ and $\Gamma \vdash_{\mathscr{C}}$ **wf** meaning that Γ is a well-formed typing context.

1.4.1 Subtyping

Definition 1.4.1
The subtyping relation induced by the cumulativity relation $\mathcal{C}_{\mathscr{C}}$ is given in Fig. 1.7. Subtyping is extended for products in a covariant way on codomains. The reasons why it is not also contravariant on domains are mostly for semantics reasons as explained in [Las12].

Remark 8 *We will show in Section 1.7.2 that the transitivity rule could be removed.*

Meta-properties of subtyping

We state here some classical properties of subtyping in CTS.

Lemma 1.4.1 *If $A \preceq_{\mathscr{C}} s$ then there exists s' such that $A \equiv_\beta s'$ and $(s', s) \in \mathcal{C}^*_{\mathscr{C}}$.*

Proof *By induction on $A \preceq_{\mathscr{C}} s$.*

Lemma 1.4.2 (Product injectivity) *If $A \preceq_{\mathscr{C}} (x:C) \to D$ then there exists C' and D' such that $A \equiv_\beta (x:C') \to D'$, $C' \equiv_\beta C$ and $D' \preceq_{\mathscr{C}} D$.*

Proof *By induction on the derivation of $A \preceq_{\mathscr{C}} (x:C) \to D$.*

Remark 9 *This lemma is also called* product compatibility *or* injectivity of product.

Subtyping is well-behaved with respect to substitution:

Lemma 1.4.3 *If $A \preceq_{\mathscr{C}} A'$ then $A\{x \leftarrow t\} \preceq_{\mathscr{C}} A'\{x \leftarrow t\}$.*

Proof *By induction on the derivation of $A \preceq_{\mathscr{C}} A'$.*

$$\frac{}{\varnothing \vdash_{\mathscr{C}} \mathbf{wf}}\ \mathscr{C}^{\mathbf{wf}}_{\varnothing}$$

$$\frac{\Gamma \vdash_{\mathscr{C}} A : s \qquad x \notin \Gamma}{\Gamma, x : A \vdash_{\mathscr{C}} \mathbf{wf}}\ \mathscr{C}^{\mathbf{wf}}_{var}$$

$$\frac{\Gamma \vdash_{\mathscr{C}} \mathbf{wf} \qquad (x : A) \in \Gamma}{\Gamma \vdash_{\mathscr{C}} x : A}\ \mathscr{C}_{var}$$

$$\frac{\Gamma \vdash_{\mathscr{C}} \mathbf{wf} \qquad (s_1, s_2) \in \mathcal{A}}{\Gamma \vdash_{\mathscr{C}} s_1 : s_2}\ \mathscr{C}_{sort}$$

$$\frac{\Gamma \vdash_{\mathscr{C}} A : s_1 \qquad \Gamma, x : A \vdash_{\mathscr{C}} B : s_2 \qquad (s_1, s_2, s_3) \in \mathcal{R}}{\Gamma \vdash_{\mathscr{C}} (x : A) \to B : s_3}\ \mathscr{C}_{\Pi}$$

$$\frac{\Gamma, x : A \vdash_{\mathscr{C}} M : B \qquad \Gamma \vdash_{\mathscr{C}} (x : A) \to B : s}{\Gamma \vdash_{\mathscr{C}} \lambda x : A.\, M : (x : A) \to B}\ \mathscr{C}_{\lambda}$$

$$\frac{\Gamma \vdash_{\mathscr{C}} M : (x : A) \to B \qquad \Gamma \vdash_{\mathscr{C}} N : A}{\Gamma \vdash_{\mathscr{C}} M\ N : B\{x \leftarrow N\}}\ \mathscr{C}_{app}$$

$$\frac{\Gamma \vdash_{\mathscr{C}} M : A \qquad \Gamma \vdash_{\mathscr{C}} B : s \qquad A \preceq_{\mathscr{C}} B}{\Gamma \vdash_{\mathscr{C}} M : B}\ \mathscr{C}_{\preceq}$$

$$\frac{\Gamma \vdash_{\mathscr{C}} M : A \qquad A \preceq_{\mathscr{C}} s}{\Gamma \vdash_{\mathscr{C}} M : s}\ \mathscr{C}^{s}_{\preceq}$$

Figure 1.8: Typing rules for CTS

1.4.2 Typing system

Definition 1.4.2 (Typing of CTS)
The typing system induced by a CTS specification $\mathscr{C}$ is defined in Fig. 1.8.

Remark 10 *When $\Gamma = \varnothing$, we say that the typing context is* closed. *This terminology is extended for judgments.*

Remark 11 *There are two typing rules for conversion ($\mathscr{C}_{\preceq}$, $\mathscr{C}^{s}_{\preceq}$) to take into account top-sorts ($s \in \mathcal{S}^{\top}_{\mathscr{C}}$). Indeed, with subtyping it is possible to have $A \preceq_{\mathscr{C}} s$ and $s \in \mathcal{S}^{\top}_{\mathscr{C}}$. Hence, we introduce the typing rule $\mathscr{C}^{s}_{\preceq}$ for this specific case.*

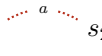
1.5 Programming with Pure Type Systems

PTS generalize many type systems that were already known before. In this section, we describe some of these systems and their related PTS specification. Names come from [AGM92]. Traditionally, sorts in PTS and CTS are denoted by $\star, \Box, \triangle, \ldots$.

Notation 8 *We use the notation λS to denote the CTS typing system induced by the specification S.*

Notation 9 *To each CTS specification, we can associate a graph. Nodes are the sorts, while arrows have the following semantics:*

- *Plain green arrows represent the relation $\mathcal{A}$. If $(s_1, s_2) \in \mathcal{A}$ then, it is picture as* 

- *Densely dotted red arrows represent the relation $\mathcal{R}$. If $(s_1, s_2, s_3) \in \mathcal{R}$ then, there are two red arrows* 

and 

where the label a is unique to each product and is here to desambiguate. Most of the time products have the form (s_1, s_2, s_2), hence only one arrow between s_1 and s_2 will be represented without label such as 

- *Dashed blue arrows represent the relation $\mathcal{C}$. If $(s_1, s_2) \in \mathcal{C}$ it will be represented as* 

Since we will always consider the transitive closure, we will only represent arrows generating the relation $\mathcal{C}$.

In the pictures below, we also do not represent all the arrows to make the graph clearer if some of them can be derived in another way. For example, if $(s_1, s_2) \in \mathcal{C}$ and $(s_2, s_2, s_2) \in \mathcal{R}$, we won't represent the arrow $(s_2, s_1, s_2) \in \mathcal{R}$ since cumulativity can always be used on the second argument. Strictly speaking, this represents two different CTS, but we will see in Chapter 2 that the two specifications are in fact equivalent. In this manuscript, this happens only for the specifications associated to the systems Coq, Lean and Matita.

Definition 1.5.1 (Simply Typed Lambda Calculus [Chu40])

$$(\to) = \begin{cases} \mathcal{S} = & \{\star, \Box\} \\ \mathcal{A} = & \{(\star, \Box)\} \\ \mathcal{R} = & \{(\star, \star, \star)\} \end{cases}$$

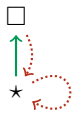
The first typed lambda-calculus invented is the Simply Typed Lambda Calculus (in 1940) and is equivalent to the CTS generated by the specification $\to$. The main property of Simply Typed Lambda Calculus is that $\hookrightarrow_\beta$ is SN. Its logical counterpart according to the proposition-as-type principle is the minimal logic (propositions are built from implications only). An example of derivable judgment in Simply Typed Lambda Calculus is given by $A : \star, B : \star, C : \star \vdash_{\to} \lambda f : A \to B \to C. \lambda a : A. \lambda b : B. f\ a\ b : C$. Simply Typed Lambda Calculus has two main limitations:

- It is not possible to form a dependent product $(x : A) \to B$ where $x \in \mathrm{FV}(B)$. Hence, through proposition-as-type principle, there is no interpretation of the $\forall$ quantifier in Simply Typed Lambda Calculus.

- It is not possible to define polymorphic functions. Hence, the identity function has to be defined as many times as it is used with different types.

Definition 1.5.2 (System *F* [Gir72] [Rey74])

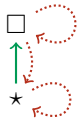
$$(\mathbf{2}) = \begin{cases} \mathcal{S} = & \{\star, \Box\} \\ \mathcal{A} = & \{(\star, \Box)\} \\ \mathcal{R} = & \{(\star,\star,\star), (\Box,\star,\star)\} \end{cases}$$



System F was invented independently by Girard and Reynolds in the 1970s, almost 30 years after the invention of Simply Typed Lambda Calculus. System F solves the two issues we raised for Simply Typed Lambda Calculus. First, this new quantification adds polymorphism allowing to express the polymorphic identity function. Indeed the following judgment is derivable: $\vdash_{\mathbf{2}} \lambda A:\star.\, \lambda a:A.\, a : (A:\star) \to A \to A$. In this judgment, A represents a *type* because a type in System F inhabits the sort $\star$. Secondly, it partially solves the first issue since its logical counterpart is the second order intuitionistic logic. This system is used as a basis for the programming language Haskell for example.

Definition 1.5.3 (System *Fω* [Gir72])

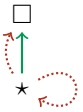
$$(\omega) = \begin{cases} \mathcal{S} = & \{\star, \Box\} \\ \mathcal{A} = & \{(\star, \Box)\} \\ \mathcal{R} = & \{(\star,\star,\star), (\Box,\star,\star), (\Box,\Box,\Box)\} \end{cases}$$



System $F\omega$ extends System F with higher-order types. A canonical example of higher-order type from the programming point of view are polymorphic `lists`. A `list` takes a type A and returns a new type: `list A`. Such construction requires to use the product $(\Box,\Box,\Box)$ to derive the judgment $A:\star \vdash_{\omega} \mathit{list}\ A : \Box$.

Definition 1.5.4 (LF [HHP93b])

$$(\mathbf{P}) = \begin{cases} \mathcal{S} = & \{\star, \Box\} \\ \mathcal{A} = & \{(\star, \Box)\} \\ \mathcal{R} = & \{(\star,\star,\star), (\star,\Box,\Box)\} \end{cases}$$



LF extends Simply Typed Lambda Calculus with dependent products (the rule $(\star,\Box,\Box)$). A canonical example of dependent product is given by *vectors*. A *vector* is a list indexed by its size. The type of (non-polymorphic) lists of length n can be represented by the type *vector* n. Indeed, one can check that the following judgment is derivable in LF: $\mathit{nat}:\star, \mathit{vect}:\mathit{nat}\to\star, n:\mathit{nat} \vdash_{\mathbf{P}} \mathit{vect}\ n : \Box$.

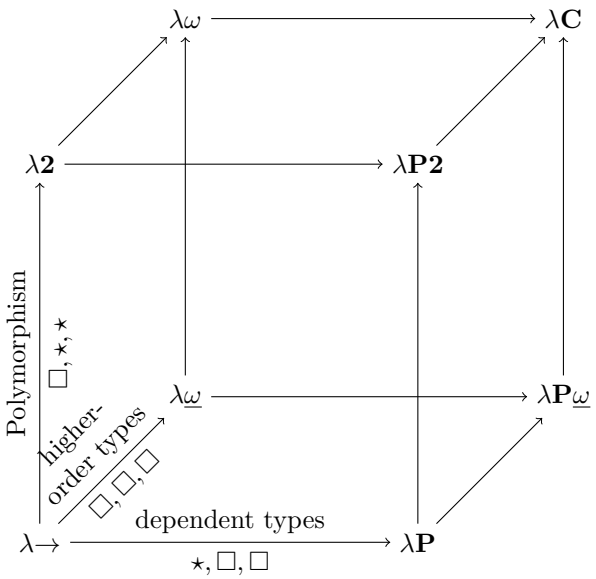
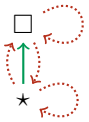


Figure 1.9: The λ-cube.

Definition 1.5.5 (Calculus of Constructions [CH86])

$$(\mathbf{C}) = \begin{cases} \mathcal{S} = & \{\star, \Box\} \\ \mathcal{A} = & \{(\star, \Box)\} \\ \mathcal{R} = & \{(\star,\star,\star), (\star,\Box,\Box), (\Box,\star,\star), (\Box,\Box,\Box)\} \end{cases}$$



Finally, we have the Calculus of Constructions which aims to gather all the features we saw previously: Simple types, dependent types, polymorphism and higher-order types. The CTS generated by this specification is quite expressive and was used as the basis for the Coq system at the end of the 1980s.

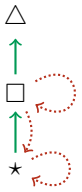
From simple types one can combine dependent types, polymorphism and Higher-Order types to generate 8 different specifications. These systems are often represented in the so-called λ-cube (or Barendregt's cube).

Definition 1.5.6 (λ-cube)
The lambda-cube represented in Fig. 1.9 is composed of eight specifications $\mathscr{P}$ such that $\mathcal{S}_{\mathscr{P}} = \{\star, \Box\}$, $\mathcal{A}_{\mathscr{P}} = \{(\star, \Box)\}$ and $\mathcal{R}_{\mathscr{P}} \subseteq \{(\star,\star,\star)\} \cup \{(i,j,j) \mid i,j \in \{\star,\Box\}\}$.

The specification we have seen so far are all member of the λ-cube. In the following, we give other famous example of PTS specifications which are not part of the lambda-cube.

Definition 1.5.7 (λHOL [Geu93])

$$(\mathbf{HOL}) = \begin{cases} \mathcal{S} = & \{\star, \Box, \triangle\} \\ \mathcal{A} = & \{(\star, \Box), (\Box, \triangle)\} \\ \mathcal{R} = & \{(\star,\star,\star), (\Box,\star,\star), (\Box,\Box,\Box)\} \end{cases}$$



This specification is a PTS version of Church's type theory [Chu40] called Simple Type Theory which gave rise to the various type systems composing the HOL family today[5] (HOL-light, HOL4, Isabelle/HOL). In this logic, $\star$ is the sort for propositions. The implication $\Rightarrow$ is encoded by the product $(\star,\star,\star)$, the forall quantifier $\forall$ is encoded by $(\Box,\star,\star)$ and function's type $\rightarrow$ is encoded by $(\Box,\Box,\Box)$. One may notice that the difference between λHOL and System $F\omega$ is only the axiom $(\Box,\triangle)$. This allows adding in a typing context type variables such as $\iota : \Box$. ι is generally used to represent natural numbers in Simple Type Theory. However, in a closed typing context, $\lambda\mathbf{HOL}$ and $\lambda\omega$ are the same since it is not possible to quantify on types that inhabit the sort $\triangle$. Another difference between the HOL family systems and System $F\omega$, is that the former is classical while the latter is intuitionistic (excluded-middle cannot be derived). In λHOL, classical logic can be added as axioms in the typing context Γ.

Non-terminating PTS:

Definition 1.5.8

$$(\star) = \begin{cases} \mathcal{S} = & \{\star\} \\ \mathcal{A} = & \{(\star,\star)\} \\ \mathcal{R} = & \{(\star,\star,\star)\} \end{cases}$$

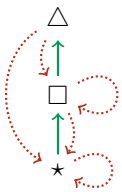
This specification is probably the simplest one we can imagine for PTS, and actually every term typable in some specification is also typable in this specification. Hence, it makes this PTS inconsistent: Through the proposition-as-type principle, one can derive a proof of False generaly represented by the proposition $(\forall A, A)$

$$\vdash_\star \lambda A:\star.\, A : (A:\star) \rightarrow A$$

: This PTS will play a role when we talk about CTS specifications embedding in Chapter 2. Adding polymorphism to λHOL gives rise to an inconsistent CTS as proved in [Coq86] [Hur95]. Polymorphism is generally added to λHOL by adding two products. This gives the system System U.

Definition 1.5.9 (System U)

$$(\mathbf{U}) = \begin{cases} \mathcal{S} = & \{\star, \Box, \triangle\} \\ \mathcal{A} = & \{(\star, \Box), (\Box, \triangle)\} \\ \mathcal{R} = & \{(\star,\star,\star), (\Box,\star,\star), (\Box,\Box,\Box), (\triangle,\star,\star), (\triangle,\Box,\Box)\} \end{cases}$$

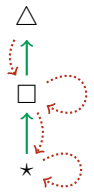


[5] These systems extend Church's simple type theory with prenex polymorphism. This extension will be discussed in Chapter. 7

Hurkens realized in [Hur95] that one product was not necessary to have a non-terminating specification. This new specification is called System U^-.

Definition 1.5.10 (System U^-)

$$(\mathbf{U}^-) = \begin{cases} \mathcal{S} = & \{\star, \Box, \triangle\} \\ \mathcal{A} = & \{(\star, \Box), (\Box, \triangle)\} \\ \mathcal{R} = & \{(\star, \star, \star), (\Box, \star, \star), (\Box, \Box, \Box), (\triangle, \Box, \Box)\} \end{cases}$$



This last specification is *minimal* in the sense that if any product or axiom is removed, then the CTS terminates. A classical interpretation for this *paradox* is that it is not possible to have impredicative universes one on top of the other. This is because in System U^-, the sorts $\star$ and $\triangle$ are impredicative.

1.5.1 Other examples of Cumulative Type Systems

So far, we have introduced systems which are PTS, that do not use the cumulativity relation on sorts. One reason for that is that most of the specifications we have introduced so far use at most three sorts. Seeing the Calculus of Constructions as a logic, $\star$ is reserved as the sort for propositions while $\Box$ is the sort for datatypes. Hence, a datatype can only inhabit in one sort, namely $\Box$. However, in practice, having only one sort for datatypes is not convenient, in particular because it is not possible to quantify over all the datatypes. This can be seen using the formalization of monoids in the Calculus of Constructions. A monoid is often represented as a record of:

- A type A of the elements (often called *carrier*)
- An inhabitant $e : A$ which is the neutral element
- An operator $\circ : A \to A \to A$
- A proof that $\forall x, x \circ e = x$
- A proof that $\forall x, e \circ x = x$
- A proof that $\forall x, \forall y, \forall z, x \circ (y \circ z) = (x \circ y) \circ z$

To simplify this example, we will omit the proofs and just stick to the computational part of the monoid. To formalize a monoid, one could define a monoid in the Calculus of Constructions as[6]:

$$(z : \star) \to ((A : \Box) \to A \to (A \to A \to A) \to z) \to z \tag{1.1}$$

This type is not valid in Calculus of Constructions, because quantifying over a type as in $(A : \Box) \to \ldots$ is not allowed in the Calculus of Constructions. Hence, it becomes very difficult to have general statements on monoids since the collection of monoids cannot be expressed is not a datatype. Instead, what is possible to do is to have general statements for monoids with a specific carrier such as $\mathbb{N}$. Indeed, the type

[6] We use a trick here to express the type of the monoid using the so-called *impredicative encoding* which is similar to encoding a datatype with its elimination principle.

$$(z : \star) \to (\mathbb{N} \to (\mathbb{N} \to \mathbb{N} \to \mathbb{N}) \to z) \to z$$

is valid in the Calculus of Constructions. A perfect solution would be to add polymorphism to Calculus of Constructions for the sort $\Box$. However, the system System U shows that adding polymorphism makes the specification inconsistant. Another idea would be to add the axiom $(\Box, \Box)$, but this makes also the logic inconsistent [MR86].

The solution adopted by many systems to solve this issue is to add a new sort $\triangle$ such that $\Box : \triangle$. Then we add the corresponding products meaning $\{(\triangle, \star, \star)\} \cup \{(i, \triangle, \triangle) \mid i \in \{\star, \Box, \triangle\}\}$. Using this new specification the type we have given in Equation 1.1 is now well-typed in $\triangle$. But this solution raises another issue which is we cannot use this definition with a carrier living in the sort $\triangle$ for the same reason as before. Hence, a natural extension to avoid this issue is to have an infinite hierarchy which extends the specification with similar rules and axioms as for $\triangle$.

Having this infinite hierarchy of universes leads to another practical issue. What should the type for the carrier of a monoid be? The sort for the carrier of a monoid is fixed, once and for all. Hence, using our specification, we would need as many datatypes for monoid as necessary (one where the carrier is $\Box$, another when the carrier is $\triangle$,...) meaning restate all the theorems about monoids at every level. To overcome this issue, there are two different solutions which require both to extend PTS. The first solution is called *universe polymorphism* [ST14]. The idea of universe polymorphism, is that given an infinite hierarchy of universes $\Box_{\mathfrak{i}}$, universe polymorphism allows you to quantify over the *level* $\mathfrak{i}$, as in $(\mathfrak{i} : \mathfrak{L}) \to \Box_{\mathfrak{i}} \to \Box_{\mathfrak{i}}$ where $\mathbb{L}$ is the type for levels. Universe polymorphism fixes the issue above by expressing the type for monoids as:

$$(\mathfrak{i} : \mathfrak{L}) \to (z : \star) \to ((A : \Box_{\mathfrak{i}}) \to A \to (A \to A \to A) \to z) \to z$$

The second solution is given with CTS by adding subtyping on sorts.

Notation 10 *Sorts–which in this context will be called universes–are denoted* $\mathfrak{0}, \mathfrak{1}, \mathfrak{2}, \ldots$.

Using cumulativity, the level for the datatype of our monoid should be the maximum needed. By needed, we mean that since every proof is finite, one should use the highest level of the datatype used as a carrier of a monoid. Hence, the type for a monoid with a carrier at level 0 and at level 100 would be the same thanks to subtyping. This gives another explanation why most of the systems based upon the Calculus of Constructions implement an infinite hierarchy of universes.

For the main systems which extend Calculus of Constructions: Agda, Coq, Lean, Matita, only Agda and Coq[7] implement universe polymorphism. The other systems as well as Coq implement a CTS with an infinite and cumulative hierarchy of universes. This makes Coq the only system to implement both universe polymorphism and cumulativity [ST14] at the time being.

In the following, we aim to give a description of the CTS behind these systems. Since we are going to have an infinite number of universes, we will represent them with numbers instead of shapes.

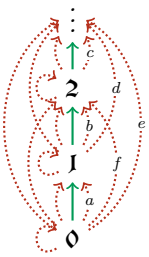
Notation 11 *For all* n*, we define* $\mathbb{N}_{<n}$ *as* $\{i \in \mathbb{N} \mid i < n\}$*. This notation is extended for* $\leq$.

[7] Since version 8.5

PTS of Agda: Since Agda has made the choice to have universe polymorphism and no cumulativity, the CTS behind Agda is in fact a PTS.

Definition 1.5.11 (Agda)
For all $n \in \mathbb{N}$, we define the class of PTS $\mathscr{P}^A_n$ which have the following specification:

$$(\mathscr{P}^A_n) = \begin{cases} \mathcal{S} = & \{\mathfrak{i} \mid \mathfrak{i} \in \mathbb{N}_{\leq n}\} \\ \mathcal{A} = & \{(\mathfrak{i}, \mathfrak{i}+1) \mid \mathfrak{i} \in \mathbb{N}_{<n}\} \\ \mathcal{R} = & \{(\mathfrak{i}, \mathfrak{j}, \mathfrak{k}) \mid \mathfrak{k} = \mathbf{max}(\mathfrak{i}, \mathfrak{j})\} \end{cases}$$



Notation 12 *We will use the notation $\mathscr{C}_\infty$ when $\mathcal{S} = \mathbb{N}$.*

The definition above introduces a family of PTS where $\mathscr{P}^A_\infty$ is actually the one behind Agda. Because of universe polymorphism, to keep the fact that every term has a type in Agda, a sort ω is added to give a type to $(\mathfrak{i} : \mathfrak{L}) \to \mathfrak{i}$ or any product which quantifies over a level.

If we compare Agda with the CTS specifications from the λ-cube, we can say that Agda has dependent types and higher-order types. However, they do not have the product associated to polymorphism. Polymorphism can be replaced with universe polymorphism. Another feature of Agda is that its specification is predicative.

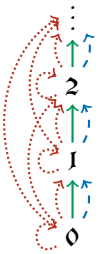
Theorem 1.5.1 *The CTS $\mathscr{P}^A_\infty$ is predicative.*

Proof *One can check that this specification is ordered using the natural order on natural numbers. Then we can prove that there is no product (s, s', s'') such that $s \triangleleft_{\mathcal{S}_{SC}} s''$ or $s' \triangleleft_{\mathcal{S}_{SC}} s''$.*

We define below an extension of Agda as a CTS by having an infinite and cumulative hierarchy of universes.

Definition 1.5.12 (Predicative cumulative hierarchy)
For all $n \in \mathbb{N}$, we define the class of CTS $\mathscr{C}_n$ which have the following specification:

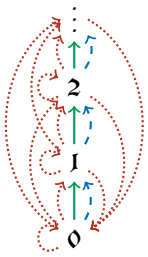
$$(\mathscr{C}_n) = \begin{cases} \mathcal{S} = & \{\mathfrak{i} \in \mathbb{N} \mid \mathfrak{i} \leq n\} \\ \mathcal{A} = & \{(\mathfrak{i}, \mathfrak{i}+1) \mid \mathfrak{i} < n\} \\ \mathcal{R} = & \{(\mathfrak{i}, \mathfrak{j}, \mathfrak{k}) \mid \mathfrak{k} = \mathbf{max}(\mathfrak{i}, \mathfrak{j})\} \\ \mathcal{C} = & \{(\mathfrak{i}, \mathfrak{j}) \mid \mathfrak{i} \leq \mathfrak{j}\} \end{cases}$$



CTS of Lean: The CTS behind Lean is very close to the PTS behind Agda. They add cumulativity on sorts and they make the sort $\mathfrak{0}$ impredicative by adding polymorphism.

Definition 1.5.13 (Lean [dMKA+15])
For all $n \in \mathbb{N}$, we define the class of CTS $\mathscr{C}^L_n$ which have the following specification:

$$(\mathscr{C}_n^L) = \begin{cases} \mathcal{S} = & \{\mathfrak{i} \in \mathbb{N}_{\leq n}\} \\ \mathcal{A} = & \{(\mathfrak{i}, \mathfrak{i}+1) \mid \mathfrak{i} \in \mathbb{N}_{<n}\} \\ \mathcal{R} = & \{(\mathfrak{i}, \mathfrak{j}, \mathfrak{k}) \mid \mathfrak{k} = \mathbf{imax}(\mathfrak{i}, \mathfrak{j})\} \\ \mathcal{C} = & \{(\mathfrak{i}, \mathfrak{j}) \mid \mathfrak{i} \leq \mathfrak{j}\} \end{cases}$$

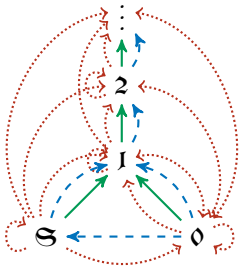


where $\mathbf{imax}(i,j)$[8] *is defined as* 0 *if* $j = 0$ *and* $\mathbf{max}(i,j)$ *otherwise.*

CTS of Coq: In Coq, they add another sort $\mathfrak{S}$, which is related to a feature of Coq called *Program Extraction* [Let08]. This means that every datatype in $\mathfrak{S}$ has a computational content that can be extracted to a programming language such as OCaml. In Coq, $\mathfrak{0}$ is used to represent propositions. This gives the following CTS.

Definition 1.5.14 (Coq [BGG+14])
For all $n \in \mathbb{N}^*$*, we define the class of CTS* $\mathscr{C}_n^C$ *which have the following specification:*

$$(\mathscr{C}_n^C) = \begin{cases} \mathcal{S} = & \{\mathfrak{i} \in \mathbb{N}_{\leq n} \cup \{\mathfrak{S}\}\} \\ \mathcal{A} = & \{(\mathfrak{i}, \mathfrak{i}+1) \mid \mathfrak{i} \in \mathbb{N}_{<n}\} \cup \{(\mathfrak{S}, 1)\} \\ \mathcal{R} = & \{(\mathfrak{i}, \mathfrak{j}, \mathfrak{k}) \mid \mathfrak{k} = \mathbf{imax}(\mathfrak{i}, \mathfrak{j})\} \\ \mathcal{C} = & \{(\mathfrak{i}, \mathfrak{j}) \mid \mathfrak{i} \leq \mathfrak{j}\} \cup \{(\mathfrak{S}, \mathfrak{i}) \cup \{\mathfrak{0}, \mathfrak{S}\} \mid \mathfrak{i} \in \mathbb{N}^{\leq n}\} \end{cases}$$



The function **imax** *is extended naturally on* $\{\mathfrak{i} \in \mathbb{N}_{\leq n} \cup \{\mathfrak{S}\}\}$ *using* $\mathfrak{S} \leq \mathfrak{1}$.

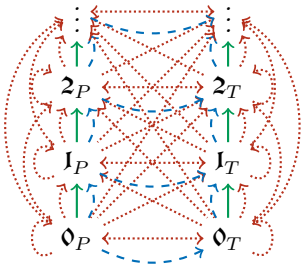
In Coq, the sort $\mathfrak{S}$ is called *Set* and $\mathfrak{0}$ is called *Prop*.

CTS of Matita: The CTS behind Matita is more complex. The idea behind this specification is to make no commitment about having an impredicative sort as in Coq or Lean. Hence, they define two predicative hierarchies of universes. There exists two mappings from one hierarchy to the other: A mapping such that all the universes are crushed into one ($\mathfrak{0}_P$, which gives $\mathscr{C}_\infty^L$), giving hence an impredicative specification. Another which maps a universe to its corresponding universe in the other hierarchy (giving the same specification as $\mathscr{C}_\infty$), hence all sorts are predicative. This notion of mapping will be formalized in Chapter 2 and are called *specification morphism.*

Definition 1.5.15 (Matita)
For all $n \in \mathbb{N}$*, we define the class of CTS* SC_n^M *which have the following specification:*

[8] for *impredicative* max

$$(\mathscr{C}_n^M) = \begin{cases} \mathcal{S} = & \{(\mathfrak{i}, t) \in \mathbb{N} \times \{P, T\} \mid \mathfrak{i} \leq n\} \\ \mathcal{A} = & \{(\mathfrak{i}, t), (\mathfrak{i}+1, t) \mid t \in \{P, T\}\} \\ \mathcal{R} = & \{(\mathfrak{i}, t'), (\mathfrak{j}, t), (\mathfrak{k}, t) \mid \mathfrak{k} = \mathbf{max}(\mathfrak{i}, \mathfrak{j})\} \\ \mathcal{C} = & \{(\mathfrak{i}, t), (\mathfrak{j}, t') \mid \mathfrak{i}, \mathfrak{j} \in \mathbb{N}, t, t' \in \{P, T\}, t \leq t', \mathfrak{i} \leq \mathfrak{j}\} \end{cases}$$



where $t \leq t'$ is defined as $\{(P,P),(P,T),(T,T)\}$.

1.6 Termination

In CTS, strongly normalization of $\hookrightarrow_\beta$ is not a syntactic property since there exists CTS that do not terminate. However, it is not as obvious as in the pure lambda-calculus since there is no type A such that the term $\lambda x : A.\, x\ x$ is typable in any CTS specification. This is a direct consequence of Product injectivity (1.4.2). We have already mentioned some CTS specifications that do not have the SN property such as $\lambda\mathbf{U}^-$.

Theorem 1.6.1 ([Hur95]) *The CTS $\lambda\mathbf{U}^-$ is not SN.*

This theorem also implies that $\lambda\mathbf{U}$ is not SN while historically, $\lambda\mathbf{U}$ was proved not SN before $\lambda\mathbf{U}^-$ by Girard [Gir72]. Non terminating CTS are not suitable in practice because they make the type checking undecidable and non suitable to define a consistent logic. Hurkens' paradox is often interpreted as being impossible to have two different impredicative sorts in the same hierarchy of universes. The PTS $\lambda\star$ is also non-terminating [MR86]. Knowing that $\lambda\mathbf{U}^-$ is non-terminating, there is an easy proof to see that $\lambda\star$ is also non-terminating. One can translate every typable judgment expressed in the $\mathbf{U}^-$ specification to a typable judgment in the $\star$ specification: This translation is the identity function except that all sorts are mapped to $\star$. Such translation between CTS specification called *sort morphisms* will be properly in Chapter 2.

On the other hand, CTS from the λ-cube are all terminating. This is implied by the termination of the Calculus of Constructions.

Theorem 1.6.2 ([GN91] [Geu94]) *The Calculus of Constructions is SN.*

Moreover, we should mention that there is a famous conjecture on PTS formulated by Barendregt [AGM92] and Geuvers [Geu93] which could be extended for CTS:

Conjecture 1 (WN implies SN) *If a PTS is WN, then it is also SN.*

This conjecture has been solved for a large class of PTS specification in [BHS01a] but remains open in the general case.

Looking at non-terminating CTS, the fact that we are not able to give a type to the following term $\lambda x : A.\, x\ x$ makes wonder if it is possible to find a CTS specification with a fixpoint combinator. This is also an open conjecture:

Conjecture 2 (Existence of fixpoint) *Is there a CTS specification which is able to type a* fixpoint *combinator? Meaning a well-typed term Y_F such that $Y_F\ F \hookrightarrow_\beta^* F\ (Y_F\ F)$ whenever $Y_F\ F$ is well-typed.*

$$\frac{}{\Gamma \vdash_{\mathscr{C}} s \ \mathbf{ws}}\ \mathbf{ws}_{\text{SORT}} \qquad \frac{\Gamma \vdash_{\mathscr{C}} A : s}{\Gamma \vdash_{\mathscr{C}} A \ \mathbf{ws}}\ \mathbf{ws}_{\text{TYPE}}$$

Figure 1.10: Derivation rules of well-sorted types

The best we can do so far, is to derive *loop* combinators: A sequence of term Y_{F_n} such that $Y_{F_n}\ F \hookrightarrow_\beta F\ (Y_{F_{n+1}}\ F)$ such that modulo a type erasure function $|\cdot|$, $|Y_{F_n}| = |Y_{F_{n+1}}|$ for all n [CH94].

1.7 Meta-theory of Cumulative Type Systems

This section states some meta-theoretical results about CTS. The main one being the subject reduction property. All the proofs can be found in [Bar99a] or [Las12]. Often, we need to say that a type A is well-sorted meaning that either A is a sort, or it has a sort. We encapsulate this definition into a judgment.

Definition 1.7.1 (Well-sorted)
*We introduce the judgment $\Gamma \vdash_{\mathscr{C}} A$ **ws** in Figure 1.10 expressing that A is well-sorted: Either A is a sort or it has a type which is a sort.*

Theorem 1.7.1 (Well-formed typing context) *If $\Gamma \vdash_{\mathscr{C}} t : A$ then $\Gamma \vdash_{\mathscr{C}}$ **wf**.*

Theorem 1.7.2 (Weakening) *If $\Gamma \vdash_{\mathscr{C}} t : A$ and $\Gamma, \Gamma' \vdash_{\mathscr{C}}$ **wf** then $\Gamma, \Gamma' \vdash_{\mathscr{C}} t : A$.*

Theorem 1.7.3 (Inversion on variable) *If $\Gamma \vdash_{\mathscr{C}} x : C$ then there exists A such that $\Gamma \vdash_{\mathscr{C}}$ **wf**, $(x : A) \in \Gamma$ and $A \preceq_{\mathscr{C}} C$.*

Theorem 1.7.4 (Inversion on sort) *If $\Gamma \vdash_{\mathscr{C}} s : C$ then there exists s' such that $\Gamma \vdash_{\mathscr{C}}$ **wf**, $(s, s') \in \mathcal{A}_{\mathscr{C}}$ and $s' \preceq_{\mathscr{C}} C$.*

Theorem 1.7.5 (Inversion on product) *If $\Gamma \vdash_{\mathscr{C}} (x : A) \to B : C$ then there exists s_1, s_2, s_3 such that $\Gamma \vdash_{\mathscr{C}} A : s_1$, $\Gamma, x : A \vdash_{\mathscr{C}} B : s_2$, $(s_1, s_2, s_3) \in \mathcal{R}_{\mathscr{C}}$ and $s_3 \preceq_{\mathscr{C}} C$.*

Theorem 1.7.6 (Inversion on abstraction) *If $\Gamma \vdash_{\mathscr{C}} \lambda x : A.\, t : C$ then there exists B and s such that $\Gamma, x : A \vdash_{\mathscr{C}} t : B$, $\Gamma \vdash_{\mathscr{C}} (x : A) \to B : s$ and $(x : A) \to B \preceq_{\mathscr{C}} C$.*

Theorem 1.7.7 (Inversion on application) *If $\Gamma \vdash_{\mathscr{C}} t\ u : C$ then there exists A, B such that $\Gamma \vdash_{\mathscr{C}} t : (x : A) \to B$, $\Gamma \vdash_{\mathscr{C}} u : A$, $B\{x \leftarrow u\} \preceq_{\mathscr{C}} A$ and $\Gamma \vdash_{\mathscr{C}} B\{x \leftarrow u\}$ **ws**.*

Theorem 1.7.8 (Substitution lemma) *If $\Gamma, x : A, \Gamma' \vdash_{\mathscr{C}} t : B$ and $\Gamma \vdash_{\mathscr{C}} N : A$ then $\Gamma, \Gamma'\{x \leftarrow N\} \vdash_{\mathscr{C}} t\{x \leftarrow N\} : B\{x \leftarrow N\}$. If $\Gamma, x : A, \Gamma' \vdash_{\mathscr{C}}$ **wf** and $\Gamma \vdash_{\mathscr{C}} N : A$ then $\Gamma, \Gamma'\{N \leftarrow A\} \vdash_{\mathscr{C}}$ **wf***

Theorem 1.7.9 (Well-sorted) *If $\Gamma \vdash_{\mathscr{C}} t : A$ then $\Gamma \vdash_{\mathscr{C}} A$ **ws**.*

Theorem 1.7.10 (Well-sorted subtyping) *If $\Gamma \vdash_{\mathscr{C}} t : A$, $\Gamma \vdash_{\mathscr{C}} B$ **ws** and $A \preceq_{\mathscr{C}} B$ then $\Gamma \vdash_{\mathscr{C}} t : B$*

Theorem 1.7.11 (Subject reduction) *If* $\Gamma \vdash_{\mathscr{C}} t : A$ *and* $t \hookrightarrow_\beta t'$ *then* $\Gamma \vdash_{\mathscr{C}} t' : A$.

Theorem 1.7.12 (Type uniqueness) *If* $\mathscr{P}$ *is a functional PTS specification,* $\Gamma \vdash_{\mathscr{P}} t : A$ *and* $\Gamma \vdash_{\mathscr{P}} t : A'$ *then* $A \equiv_\beta A'$.

η reduction with subtyping

η reduction has a strange behavior with subtyping. In particular it breaks the subject reduction property as witnessed in the following example (in the AGDA specification): In a typing context $\Gamma = f : \mathbf{2} \to \mathbf{2}$ one can derive the following judgment: $\Gamma \vdash_{\mathscr{C}} \lambda x : \mathbf{0}.\, f\ x : \mathbf{0} \to \mathbf{2}$. Even if $\lambda x : \mathbf{0}.\, f\ x \hookrightarrow_\eta f$, it is not possible to derive $\Gamma \vdash_{\mathscr{C}} f : \mathbf{0} \to \mathbf{2}$.

Such problem can be avoided (as done in the COQ system) by defining the η rewriting relation as an expansion: If $\Gamma \vdash_{\mathscr{C}} M : (x : A) \to B$, then $M \hookrightarrow_\eta \lambda x : A.\, M\ x$ and M is not a λ. This relation terminates if M is well-typed. However, this raises a practical issue: In CTS, rewriting is performed on untyped-term by computing a normal form. However, this definition of η needs to take into account the type of M to do an η-expansion. To solve this issue, the trick is to notice that in practice, reduction is always performed on well-typed terms. Therefore if one needs to check that M is convertible with $\lambda x : A.\, N\ x$, then it is sufficient to η-expand M as $\lambda x : A.\, M\ x$ since the convertibility test can assume that M and $\lambda x : A.\, N\ x$ have the same type.

Notice that having η expansion gives a form on contravariance. In the same typing context as before where $\Gamma = f : \mathbf{2} \to \mathbf{2}$, you can derive $\Gamma \vdash_{\mathscr{C}} f : \mathbf{2} \to \mathbf{2}$, but you can also derive $\Gamma \vdash_{\mathscr{C}} \lambda x : \mathbf{0}.\, f\ x : \mathbf{0} \to \mathbf{2}$ because $\mathbf{0} \preceq_{\mathscr{C}} \mathbf{2}$. So an $\eta-$expansion gives you a form of contravariant subtyping on domain for products.

1.7.1 Decidability of type-checking

The problem of the decidability of type checking is that of the decidability of the set of (Γ, t, A) such that $\Gamma \vdash_{\mathscr{C}} t : A$ is derivable. This is the fundamental problem behind any implementation of a type theory. In the case of CTS, one cannot expect that this property holds for any specification $\mathscr{C}$ for two reasons: First, the specification itself might be undecidable. For example, deciding whether $(s, s') \in \mathcal{A}_{\mathscr{C}}$ is not decidable if $\mathcal{A}_{\mathscr{C}}$ is defined as $\{(n, i) \mid \text{The } n^{th} \text{ Turing machine halts on input } i\}$. Secondly, the relation $\hookrightarrow_\beta$ might not terminate, which generally implies that $A \preceq_{\mathscr{C}} B$ is also undecidable. For these two reasons, we target CTS specifications which are decidable and normalizing for the decidability of type checking. However, even with these restrictions, it is not clear that the type checking of CTS is decidable. This is because type checking rules are not *syntax-directed*. An informal definition of syntax-directed rules taken from [Bar99b] is: A set of inference rules is said syntax-directed if using this set of rules, there is at most one way to derive a type for a given expression in a given typing context, and the type is unique. The rules which make CTS not syntax directed are $\mathscr{C}_{app}$, $\mathscr{C}_{\preceq}$ and $\mathscr{C}^s_{\preceq}$. For PTS, there are two main results about decidability of type checking. In [Jut93], it is shown that if the set of sorts is finite, then the type checking is decidable. In [Bar99b], it is shown that if the specification is functional and injective, then the type checking is also decidable. For CTS, the only result published is about semi-full CTS [Bar99a]. However, results mentioned before could probably be extended for CTS. All specifications behind concrete proof systems check one of these properties, hence type checking is decidable for these specifications.

It may be relevant to explain why the decidability of type checking is not an easy problem. Assuming assumptions we saw earlier, the difficulty about decidability of type checking in CTS comes from the combinations of the rules $\mathscr{C}_{app}$ and $\mathscr{C}_\lambda$ (this difficulty is already there for PTS). In $\mathscr{C}_{app}$, since the rule is not syntax directed, one needs to *infer* a type for the left term of the application. Hence, we need an inference algorithm which given a typing context Γ and a term

$$\frac{A{\equiv_\beta}B}{A{\preceq^t_{\mathscr{C}}}B}\preceq^t_{\equiv_\beta} \qquad \frac{(s,s')\in\mathcal{C}^*_{\mathscr{C}}}{s{\preceq^t_{\mathscr{C}}}s'}\preceq^t_{\mathcal{C}^*_{\mathscr{C}}} \qquad \frac{B{\preceq^t_{\mathscr{C}}}B'}{(x:A)\to B{\preceq^t_{\mathscr{C}}}(x:A)\to B'}\preceq^t_{\Pi}$$

Figure 1.11: CTS subtyping relation with transitivity

t returns a type A such that $\Gamma \vdash_{\mathscr{C}} t : A$ holds. One may see that defining such algorithm by induction on t is not difficult except for the case $t = \lambda x:A.\,u$, here is why. Given Γ and $\lambda x:A.\,u$, we want to infer a type C for $\lambda x:A.\,u$. By induction on the term, we can infer a type B such that $\Gamma, x:A \vdash_{\mathscr{C}} u : B$ is derivable. To conclude, we would like to prove that we can infer a sort for $(x:A)\to B$. However, we cannot use the induction hypothesis since neither $(x:A)\to B$, A or B are subterms of the original term. Allowing such recursion scheme from a term to its type is complicated. A first direction towards having such recurrence principle was done by Barthe [BHS01b] for a subclass of PTS. Also, to infer a sort to $(x:A)\to B$ we cannot simply rely on the fact that we can derive that $\Gamma \vdash_{\mathscr{C}} B$ **ws** and $\Gamma \vdash_{\mathscr{C}} A$ **ws**. The reason is without functionality, there might be several choices to give a sort s_1 to A and s_2 to B: Which one implies that there exists s_3 such that $(s_1,s_2,s_3)\in\mathcal{R}$? Other technical details are discussed in [Bar99b]. In Chapter 3 and Chapter 4, we will investigate two other directions, namely well-structured derivation trees and bi-directional CTS, to solve that kind of issue which arises for many other problems related to CTS. In particular, this problem also appears in a different way during the soundness proof of our encoding of CTS into λΠ-CALCULUS MODULO THEORY.

1.7.2 Subtyping and transitivity

The transitivity rule for subtyping is an issue to implement a type checker. Indeed it is not structural and therefore it is not clear when to use it. The way it is achieved in modern proof assistants is to check first the convertibility and if they are not convertible reduce the types until we found a sort or a product. If both types are sorts, subtyping is checked via the specification directly. If both types are product we check the convertibility of domains and we apply recursively on co-domains.. In this section we formalize this idea by defining another subtyping relation without the transitivity rule presented in Fig 1.11. Accordingly, we define $\Gamma \vdash^t_{\mathscr{C}} t : A$ and $\Gamma \vdash^t_{\mathscr{C}}$ **wf** the new type system which uses $\preceq^t_{\mathscr{C}}$ for the subtyping relation. The point of this section is to show that the two type systems defined by these relations are equivalent even though the two subtyping relations are different.

It is clear that if $\Gamma \vdash^t_{\mathscr{C}} t : A$ then $\Gamma \vdash_{\mathscr{C}} t : A$ and if $\Gamma \vdash^t_{\mathscr{C}}$ **wf** then we have $\Gamma \vdash_{\mathscr{C}}$ **wf**. But what about the opposite direction? In that case, the idea to simulate transitivity of the cumulativity relation is to stack several applications of $\mathscr{C}_{\preceq}$ or $\mathscr{C}^s_{\preceq}$. The difficulty comes from that in $\Gamma \vdash^t_{\mathscr{C}} t : A$ all the intermediate types needs to be well-typed which is not the case with the subtyping rule .

To resolve this issue we introduce in Fig 1.12 an intermediate cumulativity relation $\preceq^{t^-}_{\mathscr{C}}$ where the transitivity rule is admissible. From this cumulativity relation we will be able to derive that every intermediate steps are typable.

First some technical lemmas that shows that $\preceq^{t^-}_{\mathscr{C}}$ is closed with conversion.

Lemma 1.7.13 *If $A{\preceq^{t^-}_{\mathscr{C}}}B$ and $B{\equiv_\beta}C$ then $A{\preceq^{t^-}_{\mathscr{C}}}C$*

Proof *By induction on $A{\preceq^{t^-}_{\mathscr{C}}}B$. All the cases are trivial by transitivity of $\equiv_\beta$.*

$$\frac{A \equiv_\beta B}{A \preceq^{t^-}_{\mathscr{C}} B}\ \preceq^{t^-}_{\equiv_\beta} \qquad \frac{A \equiv_\beta s \qquad B \equiv_\beta s' \qquad (s,s') \in \mathcal{C}^*_{\mathscr{C}}}{A \preceq^{t^-}_{\mathscr{C}} B}\ \preceq^{t^-}_{\mathcal{C}^*_{\mathscr{C}}}$$

$$\frac{A \equiv_\beta (x:A_1) \to A_2 \qquad B \equiv_\beta (x:B_1) \to B_2 \qquad A_1 \equiv_\beta B_1 \qquad A_2 \preceq^{t^-}_{\mathscr{C}} B_2}{A \preceq^{t^-}_{\mathscr{C}} B}\ \preceq^{t^-}_{\Pi}$$

Figure 1.12: CTS subtyping relation where transitivity is admisslbe

Lemma 1.7.14 *If $B \preceq^{t^-}_{\mathscr{C}} C$ and $A \equiv_\beta B$ then $A \preceq^{t^-}_{\mathscr{C}} C$*

Proof *By induction on $B \preceq^{t^-}_{\mathscr{C}} C$. All the cases are trivial by transitivity of $\equiv_\beta$.*

Hence, we can simulate transitivity in $\preceq^{t^-}_{\mathscr{C}}$.

Lemma 1.7.15 (Transitivity of $\preceq^{t^-}_{\mathscr{C}}$) *If $A \preceq^{t^-}_{\mathscr{C}} B$ and $B \preceq^{t^-}_{\mathscr{C}} C$ then $A \preceq^{t^-}_{\mathscr{C}} C$.*

Proof *To get the good induction hypothesis we need first, to generalize over $B \preceq^{t^-}_{\mathscr{C}} C$ and C, and then by induction on $A \preceq^{t^-}_{\mathscr{C}} B$. Finally, by an inversion on $B \preceq^{t^-}_{\mathscr{C}} C$. Among the 9 cases possible, they all can be closed easily except when the last rule is $\preceq^{t^-}_{\Pi}$ in both cases. In that case we emphasize that the induction hypothesis is the following one: If $A_2 \preceq^{t^-}_{\mathscr{C}} B_2$ then for all C such that $B_2 \preceq^{t}_{\mathscr{C}} C$ we have $A_2 \preceq^{t^-}_{\mathscr{C}} C$.*

$\Diamond$ $\preceq^{t^-}_{\Pi}, \preceq^{t^-}_{\Pi}$:

	(1)	$A \preceq^{t^-}_{\mathscr{C}} B$	*Main hypothesis*	
	(2)	$B \preceq^{t^-}_{\mathscr{C}} C$		
	(3)	$A \equiv_\beta (x:A_1) \to A_2$	*Inversion on* $\preceq^{t^-}_{\Pi}$	*1*
	(4)	$B \equiv_\beta (x:B_1) \to B_2$		
	(5)	$A_1 \equiv_\beta B_1$		
	(6)	$A_2 \preceq^{t^-}_{\mathscr{C}} B_2$		
	(7)	$B \equiv_\beta (x:B_3) \to B_4$	*Inversion on* $\preceq^{t^-}_{\Pi}$	*2*
	(8)	$C \equiv_\beta (x:C_1) \to C_2$		
	(9)	$B_3 \equiv_\beta C_1$		
	(10)	$B_4 \preceq^{t^-}_{\mathscr{C}} C_2$		
	(11)	$B_1 \equiv_\beta B_3$	*Product injectivity (1.4.2)*	*4,7*
	(12)	$B_2 \equiv_\beta B_4$		
	(13)	$A_1 \equiv_\beta C_1$	*Transitivity of* $\equiv_\beta$	*5,11,9*
	(14)	$B_2 \preceq^{t^-}_{\mathscr{C}} C_2$	*Lemma (1.7.14)*	*6,12*
	(15)	$A_2 \preceq^{t^-}_{\mathscr{C}} C_2$	*Induction Hypothesis*	*6,14*
$\bigstar$	(16)	$A \preceq^{t^-}_{\mathscr{C}} B$	$\preceq^{t^-}_{\Pi}$	*3,8,13,15*

Lemma 1.7.16 *If $A \preceq_{\mathscr{C}} B$ then $A \preceq^{t^-}_{\mathscr{C}} B$.*

Proof *By induction on $A \preceq_{\mathscr{C}} B$.*

◊ $\preceq_{\equiv_\beta}$:

Trivial by $\preceq^{t^-}_{\equiv_\beta}$

◊ $\preceq_{\mathcal{C}^*_{\mathscr{C}}}$:

Trivial by $\preceq^{t^-}_{\mathcal{C}^*_{\mathscr{C}}}$

◊ $\preceq_{\Pi}$:

Trivial by $\preceq^{t^-}_{\Pi}$ *using induction hypothesis.*

◊ $\preceq_{trans}$:

By induction hypothesis using Lemma 1.7.15.

Using the relation $\preceq^{t^-}_{\mathscr{C}}$ we were able to remove the transitivity rule by inlining conversions. We now need the following intermediate lemma:

Lemma 1.7.17 *If* $A \preceq^{t^-}_{\mathscr{C}} B$ *then there exists* A' *and* B' *such that* $A' \preceq^{t}_{\mathscr{C}} B'$ *with* $A \hookrightarrow^*_\beta A'$ *and* $B \hookrightarrow^*_\beta B'$.

Proof *By induction on* $A \preceq^{t^-}_{\mathscr{C}} B$.

◊ $\preceq^{t^-}_{\equiv_\beta}$:

Trivial by $\preceq^{t}_{\equiv_\beta}$ *by taking* $A' = A$ *and* $B' = B$.

◊ $\preceq^{t^-}_{\mathcal{C}^*_{\mathscr{C}}}$:

	(1)	$A \preceq^{t^-}_{\mathscr{C}} B$	*Main hypothesis*	
	(2)	$A \equiv_\beta s$	*Inversion on* $\preceq^{t^-}_{\mathcal{C}^*_{\mathscr{C}}}$	1
	(3)	$B \equiv_\beta s'$		
	(4)	$(s, s') \in \mathcal{C}^*_{\mathscr{C}}$		
	(5)	$A \hookrightarrow^*_\beta s$	*Confluence of* β	2
	(6)	$B \hookrightarrow^*_\beta s'$	*Confluence of* β	3
	(7)	$s \preceq^{t}_{\mathscr{C}} s'$	$\preceq^{t}_{\mathcal{C}^*_{\mathscr{C}}}$	4
	(8)	*Let* $A' := s$		
	(9)	*Let* $B' := s'$		
✯	(10)	$A' \preceq^{t}_{\mathscr{C}} B'$	*Definition of* A' *and* B'	7,8,9
✯	(11)	$A \hookrightarrow^*_\beta A'$	*Definition of* A'	5,8
✯	(12)	$B \hookrightarrow^*_\beta B'$	*Definition of* B'	6,9

◊ $\preceq^{t^-}_{\Pi}$:

	(1)	$A \preceq^{t^-}_{\mathscr{C}} B$	*Main hypothesis*	
	(2)	$A \equiv_\beta (x:A_1) \to A_2$	*Inversion on* $\preceq^{t^-}_{\Pi}$	*1*
	(3)	$B \equiv_\beta (x:B_1) \to B_2$		
	(4)	$A_1 \equiv_\beta B_1$		
	(5)	$A_2 \preceq^{t^-}_{\mathscr{C}} B_2$		
	(6)	$D \preceq^{t}_{\mathscr{C}} E$	*Induction Hypothesis*	*5*
	(7)	$A_2 \hookrightarrow^*_\beta D$		
	(8)	$B_2 \hookrightarrow^*_\beta E$		
	(9)	$A_1 \hookrightarrow^*_\beta C \hookleftarrow_\beta B_1$	*Confluence of* β	*4*
	(10)	$(x:C) \to D \preceq_{\mathscr{C}} (x:C) \to E$	$\preceq^t_\Pi$	*6*
	(11)	*Let* $A' = (x:C) \to D$		
	(12)	*Let* $B' = (x:C) \to E$		
★	(13)	$A' \preceq^t_{\mathscr{C}} B'$	*Definition of* A' *and* B'	*10,11,12*
★	(14)	$A \hookrightarrow^*_\beta A'$	*Congruence of* β	*11,7,9*
★	(15)	$B \hookrightarrow^*_\beta B'$	*Congruence of* β	*12,8, 9*

Lemma 1.7.18 *If* $\Gamma \vdash^t_{\mathscr{C}} t : A$, $\Gamma \vdash_{\mathscr{C}} B$ **ws** *and* $A \preceq^{t^-}_{\mathscr{C}} B$ *then* $\Gamma \vdash^t_{\mathscr{C}} t : B$.

Proof

	(1)	$\Gamma \vdash_{\mathscr{C}} t : A$	*Main hypothesis*	
	(2)	$\Gamma \vdash^t_{\mathscr{C}} B$ **ws**		
	(3)	$A \preceq^{t^-}_{\mathscr{C}} B$		
	(4)	$A' \preceq^t_{\mathscr{C}} B'$	*Lemma (1.7.17)*	*3*
	(5)	$A \hookrightarrow^*_\beta A'$		
	(6)	$B \hookrightarrow^*_\beta B'$		
	(7)	$\Gamma \vdash^t_{\mathscr{C}} A$ **ws**	*Well-sorted*	*1*
	(8)	$\Gamma \vdash^t_{\mathscr{C}} A'$ **ws**	*Subject reduction*	*7,5*
	(9)	$\Gamma \vdash^t_{\mathscr{C}} B'$ **ws**	*Subject reduction*	*2,6*
	(10)	$A \preceq_{\mathscr{C}} A'$	$\preceq^t_{\equiv_\beta}$	*5*
	(11)	$B' \preceq_{\mathscr{C}} B$	$\preceq^t_{\equiv_\beta}$	*6*
	(12)	$\Gamma \vdash^t_{\mathscr{C}} t : A'$	*Well-sorted subtyping*	*1,8,10*
	(13)	$\Gamma \vdash^t_{\mathscr{C}} t : B'$	*Well-sorted subtyping*	*12,9,4*
★	(14)	$\Gamma \vdash^t_{\mathscr{C}} t : B$	*Well-sorted subtyping*	*13,2,11*

We can now finally conclude:

Lemma 1.7.19 *We have the following implications:*

- $\Gamma \vdash_{\mathscr{C}} t : A$ *then* $\Gamma \vdash^t_{\mathscr{C}} t : A$
- $\Gamma \vdash_{\mathscr{C}}$ **wf** *then* $\Gamma \vdash^t_{\mathscr{C}}$ **wf**

Proof *By induction on the derivation of* $\Gamma \vdash_{\mathscr{C}} t : A$ *and* $\Gamma \vdash_{\mathscr{C}}$ **wf**. *The subtyping rules are handled using Lemma 1.7.16 and Lemma 1.7.18.*

Related work: It is interesting to note that for subtyping, Marc Lasson [Las12] used $\preceq^{t^-}_{\mathscr{C}}$ while Ali Assaf [Ass15b] used $\preceq_{\mathscr{C}}$. However Marc Lasson had a subtle difference where he did not considered the transitivity of the cumulativity relation in the rule $\preceq_{\mathcal{C}^*_{\mathscr{C}}}$. He showed that in that case, the two definitions are different. This is not surprising and will be a consequence of results developed in Section 2.2.2. In particular, this difference is subsumed by the notion of weak equivalence between CTS that we develop in Chapter 2.

This equivalence is actually a simpler case of semantics CTS which are developed in Chapter 3 (which extend semantics PTS introduced by Geuvers [Geu93]). In semantics CTS, every β steps needs to be well-typed. However, by doing so, we do not know how to prove subject reduction anymore on this new system because we lose the Product injectivity (1.4.2) property. We will developed a technique in Chapter 3 which aims to solve this difficult question.

Chapter 2

Embeddings of CTS specifications

Interoperability between proof assistants requires first to understand how proofs from one logic can be translated into another. In Chapter 1, we have explained how CTS provide a theoretical basis behind many concrete systems. Understanding how proofs can be translated from one of these systems to another require first to understand how one can translate proofs from one CTS[1] to another. In this chapter we will develop three different definitions of embedding. All these equivalences will be used throughout the manuscript and this is why we have decided to present these equivalences one by one from the strongest to the weakest.

A notion that existed first for PTS and can be extended for CTS is to define a morphism of sorts which is compatible with $\mathcal{A}, \mathcal{R}$ and $\mathcal{C}$ called *specification morphism*. However, this idea is often too strong and cannot be used in practice for two reasons:

- In general, we are not interested in a total translation from a CTS to another, but only a partial one that can be used effectively. For example, the CTS behind Matita allows an arbitrary number of universes while the CTS behind the HOL family systems allows only three universes. We will see in Chapter 11, that in practice, many proofs developed in Matita, especially arithmetic proofs do not use all the expressivity of Matita's type system and as a consequence, these proofs can be translated to the HOL family systems,
- Given the position of a sort inside a judgment, we may translate this sort differently (depending if it is seen as a sort, a type or a term for example) which is not possible with a specification morphism.

Hence, we weaken the notion of specification morphism to have a more general definition of enbedding called *CTS embedding*. Using this new definition of embedding, we can derive general results about equivalences between CTS. In particular, we will show that:

- Any CTS is equivalent to a functional CTS (Definition 1.3.6)
- Any CTS is equivalent to an injective CTS (Definition 1.3.7)

We also introduce a weaker version of embedding called *weak CTS embedding* when a judgment is translated to an equivalent judgment modulo a substitution. In particular, we will show that:

- Any CTS is weakly-equivalent to a CTS with at most one top-sort

[1] Formally the translations are between CTS specifications

- Any CTS is weakly-equivalent to a CTS without top-sorts

Weak CTS equivalence and CTS equivalence coincide for judgments with an closed typing context. We think that these equivalences can be used in a broader perspective, for example to decide the type checking of a large class of CTS.

In a second part of this chapter, we are looking for an effective procedure to know whether a judgment can be translated from a CTS $\mathscr{C}$ into a CTS $\mathscr{C}'$. The notion of embedding we use for this procedure is the second one: CTS embedding. The idea is to define a notion of free CTS associated to a derivation of a judgment $\Gamma \vdash_{\mathscr{D}} t : A$. Then, knowing if a judgment can be embedded into $\mathscr{C}$ is the same as finding a specification morphism from this free CTS to $\mathscr{C}$. While in general, the process of finding such specification morphism is not decidable, we will see that in practice, it is in fact decidable. We have remarked while writing this manuscript that some of these ideas were already present in a workshop paper of Randy Pollack [Pol92] while he was fixing a result about the type checking in PTS. Free CTS can also be seen as a generalization of the cycle-detection algorithm implemented for Coq [GJCP19].

We conclude this chapter with a discussion about the completeness of our method. The main issue is that our method depends on some derivation tree for a judgment. Hence, if our method fails, it only means that the derivation tree built for this judgment cannot be embedded into the CTS $\mathscr{C}$. This does not allow us to conclude that the judgment itself cannot be embedded into $\mathscr{C}$. Getting completeness requires to build a so-called *canonical tree*. Roughly, the idea is the free CTS of a canonical tree should be more general than the free CTS of any other derivation trees. However, the existence of such canonical tree is left as a conjecture.

2.1 Equivalences between CTS

This section aims at giving tools to talk about interoperability between CTS specifications. We will define three different notions of embeddings between CTS in this section, namely: specification morphism $\trianglelefteq_\sigma$ (2.1.1), CTS embedding ($\trianglelefteq$) (2.1.5) and weak CTS embedding ($\trianglelefteq^w$) (2.1.12). All these notions of embeddings can be ordered by inclusion as follows:

$$\trianglelefteq\sigma \subset \trianglelefteq \subset \trianglelefteq^w$$

All these notions of embeddings naturally define also a notion of equivalence. The property we aim to have for these equivalences is the preservation of termination: If $\mathscr{C}$ is a terminating CTS specification and is equivalent to another CTS specification $\mathscr{D}$, then $\mathscr{D}$ is also a terminating specification.

We take time to introduce these definitions one by one because we will mention all of them in the remaining part of the manuscript. However, in practice we are mainly interested to the sechond notion of embedding ($\trianglelefteq$). It is the one for which we have implemented a decision procedure which is explained in Section 2.3 and its implementation is the object of Chapter 10.

2.1.1 Specification morphisms

A first idea about interoperability between CTS would be the following: Given two CTS specifications $\mathscr{C}$ and $\mathscr{D}$ as well as a derivable judgment $\Gamma \vdash_{\mathscr{C}} t : A$, is it possible to derive $\Gamma \vdash_{\mathscr{D}} t : A$? The question does not make sense as, when $\mathcal{S}_{\mathscr{C}} \neq \mathcal{S}_{\mathscr{D}}$, the language of $\mathscr{C}$ and $\mathscr{D}$ are also different. For this reason, we need to introduce a function between sorts $\sigma : \mathcal{S}_{\mathscr{C}} \to \mathcal{S}_{\mathscr{D}}$. We can refine our first idea using the function σ to rephrase the question to say that if $\Gamma \vdash_{\mathscr{C}} t : A$ is derivable, is it possible to derive $\Gamma\sigma \vdash_{\mathscr{D}} t\sigma : A\sigma$ where σ is extended naturally on terms and typing contexts. To build a derivation tree for this judgment, we expect σ to be compatible with $\mathcal{A}_{\mathscr{D}}$, $\mathcal{R}_{\mathscr{D}}$ and $\mathcal{C}_{\mathscr{D}}$. This intuition is captured by the definition of *specification morphism*.

Definition 2.1.1 (Specification morphism)
Let $\mathscr{C}$ and $\mathscr{D}$ be two CTS specifications. We say that $\sigma : \mathcal{S}_\mathscr{C} \to \mathcal{S}_\mathscr{D}$ is a specification morphism *if:*

$$(s, s') \in \mathcal{A}_\mathscr{C} \Rightarrow (\sigma(s), \sigma(s')) \in \mathcal{A}_\mathscr{D}$$
$$(s, s', s'') \in \mathcal{R}_\mathscr{C} \Rightarrow (\sigma(s), \sigma(s'), \sigma(s'')) \in \mathcal{R}_\mathscr{D}$$
$$(s, s') \in \mathcal{C}_\mathscr{C} \Rightarrow (\sigma(s), \sigma(s')) \in \mathcal{C}_\mathscr{D}$$

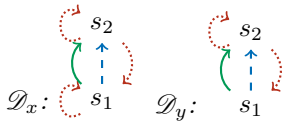
We will also denote $\mathscr{C} \trianglelefteq_\sigma \mathscr{D}$ the specification morphism $\sigma : \mathcal{S}_\mathscr{C} \to \mathcal{S}_\mathscr{D}$. Specification morphism is extended naturally on terms and typing contexts.

Theorem 2.1.1 (Morphism soundness) *If $\sigma : \mathcal{S}_\mathscr{C} \to \mathcal{S}_\mathscr{D}$ is a specification morphism then if $\Gamma \vdash_\mathscr{C} t : A$ is derivable, then so is $\Gamma\sigma \vdash_\mathscr{D} t\sigma : A\sigma$.*

Proof *By induction on the derivation of $\Gamma \vdash_\mathscr{C} t : A$.*

We give in Example 2.1 and 2.2 two limitations of sort-morphism.

Example 2.1 *We denote $\mathscr{D}_x$ and $\mathscr{D}_y$ the specifications given by the following graphs (graphs of CTS are defined in Definition 9):*



The only difference between $\mathscr{D}_x$ and $\mathscr{D}_y$ is the product (s_1, s_1, s_1). However, using cumulativity, this product can be simulated in $\mathscr{D}_y$. Assuming we have $\Gamma \vdash_{\mathscr{D}_y} t : s_1$ and $\Gamma \vdash_{\mathscr{D}_y} u : s_1$. Then we can derive $\Gamma \vdash_{\mathscr{D}_y} t \to u : s_1$ as follows:

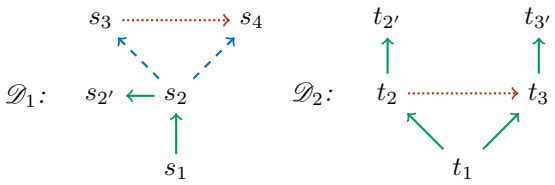
$$\dfrac{\dfrac{\Gamma \vdash_{\mathscr{D}_y} t : s_1 \qquad (s_1, s_2) \in \mathcal{C}_{\mathscr{D}_y}}{\Gamma \vdash_{\mathscr{D}_y} t : s_2} \qquad \dfrac{}{\Gamma \vdash_{\mathscr{D}_y} u : s_1} \qquad (s_2, s_1, s_1) \in \mathcal{R}_{\mathscr{D}_y}}{\Gamma \vdash_{\mathscr{D}_y} t \to u : s_1}$$

Hence, even if there is no specification morphism from $\mathscr{D}_x$ to $\mathscr{D}_y$, any derivable judgment in $\mathscr{D}_x$ can also be derived in $\mathscr{D}_y$.

Example 2.2 *Obviously, there is no specification morphism between the* Calculus of Constructions *with 5 and 2 universes ($\mathscr{C}_5$ and $\mathscr{C}_2$ (Definition 1.5.12)), however the judgment $\vdash_{\mathscr{C}_5} \mathfrak{O} : \mathfrak{I}$ can be derived in $\mathscr{C}_2$.*

For this last example, one could relax the definition of specification morphism so that σ is only a partial function, however this is still not enough because of the first example. In general, the position of a sort inside the judgment may change its translation. To make this idea more precise, we give another example below. The specifications $\mathscr{D}_1$ and $\mathscr{D}_2$ will be reused throughout this chapter.

Example 2.3 *We denote $\mathscr{D}_1$ and $\mathscr{D}_2$ the specifications given by the following graphs*



One can check that the following judgment is derivable $x : s_2, y : s_2 \vdash_{\mathscr{D}_1} x \to y : s_4$. *However, there is no specification morphism which makes this judgment derivable in* $\mathscr{D}_2$. *This is simply because* s_2 *cannot be mapped to a sort so that the product* $x \to y$ *is well-typed. Indeed, using specification morphisms, we cannot map the first occurrence of* s_2 *to* t_2 *and the second occurrence to* t_3 *to produce the judgment* $x : t_2, y : t_3 \vdash_{\mathscr{D}_2} x \to y : t_3$. *Moreover, we can notice that there is no specification morphism from* $\mathscr{D}_2$ *to* $\mathscr{D}_1$. *Still, we will see later that these two specifications are equivalent using the notion of* weak CTS equivalence *(Definition 2.1.12).*

All the examples above make it clear that specification morphism is most of the time too restrictive to define an interesting equivalence relation between CTS specifications. We define a weaker notion of equivalence in the next section called *CTS embedding*.

2.1.2 CTS embeddings

Our notion of CTS embedding generalizes specification morphisms. Because of the sorts, the syntax between two CTS might be different. Hence, we are interested in having a notion of equality between terms and judgment which does not depend on the sorts anymore. Our idea is to use the $\star$ specification (Definition 1.5.8) which has only one sort. Morever, this specification has the the property that from any CTS specification, there is only one canonical specification morphism to $\star$.

Theorem 2.1.2 *For any CTS specification* $\mathscr{C}$, *there is a canonical specification morphism to* $\star$.

Proof *Every sort is mapped to* $\star$.

Definition 2.1.2 (Sort erasure)
We will use the notation $t_\star$ *to denote* $t\sigma$ *where* σ *is the specification morphism defined in Theorem 2.1.2. We say that* $t_\star$ *is the* sort-erasure *of* t. *This notation is extended to typing contexts, judgments and derivation trees.*

Notation 13 *We define the notation* $=_\star$ *as:* $t =_\star t' := t_\star = t'_\star$. *This equality is extended naturally to typing contexts, judgments and derivation trees.*

Lemma 2.1.3 *If* $t =_\star t'$ *and* $t \hookrightarrow_\beta t_1$, *then there exists* $t_{1'}$ *such that* $t_1 \hookrightarrow_\beta t_{1'}$ *and* $t_1 =_\star t_{1'}$.

Proof *Every* β*-redex in* t *is also a* β*-redex in* t'.

Definition 2.1.3 (Judgment $\star$-embedding)
For any specification, we say that the judgment $\Gamma \vdash_\star t : A$ *is* $(\star, \mathscr{C})$*-embedded if there exist* Γ', t', A' *such that* $\Gamma' \vdash_{\mathscr{C}} t' : A'$ *is derivable with* $\Gamma =_\star \Gamma'$, $t =_\star t'$ *and* $A =_\star A'$.

Definition 2.1.4 (Judgment embedding)
For any CTS specification $\mathscr{C}$ *and* $\mathscr{C}$*', a judgment* $\Gamma \vdash_{\mathscr{C}} t : A$ *is* $(\mathscr{C}, \mathscr{C}')$*-embedded if the judgment* $\Gamma_\star \vdash_\star t_\star : A_\star$ *is* $(\star, \mathscr{C}')$*-embedded.*

Example 2.4 *Using again the specifications $\mathscr{D}_1$ and $\mathscr{D}_2$ from Example 2.3, one can check that the judgment $x : s_2, y : s_2 \vdash_{\mathscr{D}_1} x \to y : s_4$ is $(\mathscr{D}_1, \mathscr{D}_2)$-embedded. Indeed, the judgment $x : t_2, y : t_3 \vdash_{\mathscr{D}_2} x \to y : t_3$ is derivable and their sort-erasure is the same.*

Definition 2.1.5 (CTS embedding & CTS equivalence)
For any two CTS specifications $\mathscr{C}$ and $\mathscr{C}'$, we say that $\mathscr{C}$ is CTS embedded *into $\mathscr{C}'$ if any derivable judgment $\Gamma \vdash_{\mathscr{C}} t : A$ is $(\mathscr{C}, \mathscr{C}')$-embedded. Two CTS are said* CTS equivalent *if one is CTS embedded into the other and vice versa.*

Notation 14 *We write $\mathscr{C} \trianglelefteq \mathscr{C}'$ to express that $\mathscr{C}$ is CTS embedded into $\mathscr{C}'$. We write $\mathscr{C} \sim \mathscr{C}'$ to express that the two specifications are CTS equivalent.*

The Example 2.1 put in evidence that there are CTS specifications which are minimal in the sense that any axiom (resp. rule) cannot be *simulated* from other axioms (resp. rules) using cumulativity. This is specified in the definition below.

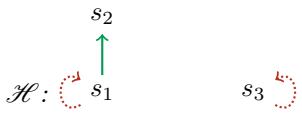
Definition 2.1.6 (Minimal specification [Ass15b])
Given a specification $\mathscr{C}$, we define the minimal specification $\mathscr{C}^{\mathcal{M}}$ as follows:

- $\mathcal{S}_{\mathscr{C}^{\mathcal{M}}} = \mathcal{S}_{\mathscr{C}}$
- $\mathcal{A}_{\mathscr{C}^{\mathcal{M}}} = \{(s_1, s_2) \mid \forall s_{2'} \in \mathcal{S}_{\mathscr{C}}, s_2 \neq s_{2'} \wedge (s_1, s_{2'}) \in \mathcal{A}_{\mathscr{C}} \Rightarrow (s_{2'}, s_2) \notin \mathcal{C}^*_{\mathscr{C}}\}$
- $\mathcal{R}_{\mathscr{C}^{\mathcal{M}}} = \{(s_1, s_2, s_3) \mid \forall s_{1'}, s_{2'}, s_{3'}, (s_1, s_2, s_3) \neq (s_{1'}, s_{2'}, s_{3'}) \wedge (s_{1'}, s_{2'}, s_{3'}) \in \mathcal{R}_{\mathscr{C}} \Rightarrow \{(s_1, s_{1'}), (s_2, s_{2'}), (s_{3'}, s_3)\} \not\subseteq \mathcal{C}^*_{\mathscr{C}}\}$
- $\mathcal{C}_{\mathscr{C}^{\mathcal{M}}} = \{(s_1, s_2) \mid \forall s_{2'} \in \mathcal{S}_{\mathscr{C}}, s_2 \neq s_{2'} \wedge (s_1, s_{2'}) \in \mathcal{C}_{\mathscr{C}} \Rightarrow (s_{2'}, s_2) \notin \mathcal{C}^*_{\mathscr{C}}\}$

Theorem 2.1.4 ([Ass15b]) *For any specification $\mathscr{C}$, we have ($\mathscr{C}^{\mathcal{M}} \sim \mathscr{C}$)*

Proof *This proof is a direct consequence of the definition of CTS judgment equivalence and the definition of $\mathscr{C}^{\mathcal{M}}$.*

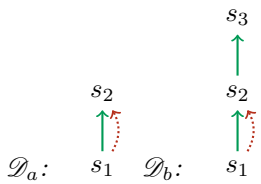
Remark 12 *The notion of* minimality *defined above is related to cumulativity as shown in the example below.*



The product (s_3, s_3, s_3) cannot be used and as such the specification $\mathscr{H}$ is equivalent to the Simply Typed Lambda Calculus. *Defining a proper definition of minimality requires to know whether a sort is inhabited (see in Section 2.1.4).*

The notion of embedding we have defined is the one we will use in practice and it is the one for which we will define a decidable procedure in Section 2.3. However, this equivalence does not behave well with top-sorts and cumulativity.

Example 2.5 *We denote $\mathscr{D}_a$ and $\mathscr{D}_b$ the specifications given by the following graphs:*



Without the product (s_1, s_2, s_2) it is not hard to see that the two specifications would be equivalent using CTS embeddings. However, using this product, one can construct the judgment $X : s_1, Y : s_2 \vdash_{\mathscr{D}_b} X \to Y : s_2$ but it has no equivalent in $\mathscr{D}_a$ using the notion of CTS embedding. The reason is because the variable $Y : s_2$ cannot be declared in $\mathscr{D}_a$ because s_2 is a top-sort and hence s_2 has no type (see $\mathscr{C}_{var}$).

A kind of equivalent judgment derivable in $\mathscr{D}_a$ would be $X : s_1 \vdash_{\mathscr{D}_a} X \to s_1 : s_2$. In this new judgment, we have substituted s_1 for the variable Y, an inhabitant of s_2. It is equivalent in a sense that we have replaced a variable of some type by a term of the same type. Indeed, any variable of type s_2 in $\mathscr{D}_b$ can only appear in a typing context because there is no product which start with s_3, the type of s_2.

The next notion of embedding we will define, aims at fixing the issue raised by the example above with a weaker notion of embedding where we relax CTS equivalence modulo a substitution. We highlight at the end of this part (see Section 7.4) that this problem of cumulativity and top-sorts maybe actually a problem of the definition of CTS because top-sorts are not *types* because a variable cannot inhabit a top-sort.

2.1.3 Weak CTS embeddings

The Example 2.5 shows an issue with CTS equivalence related to top-sorts. Extending a CTS specification with an axiom (s_1, s_2) where s_1 is a top-sort allows to declare variable of type s_1 which was not possible before. If s_1 was inhabited, meaning that there exist Γ and t such that $\Gamma \vdash_{\mathscr{C}} t : s_1$, we can always replace any variable which inhabit s_1 by t. Adding this axiom does not allows to inhabit more types.

2.1.4 Inhabitation of top-sorts

Definition 2.1.7 (Inhabitation of a type)
A type A in a specification $\mathscr{C}$ is inhabited *if there exist Γ and t such that $\Gamma \vdash_{\mathscr{C}} t : A$. Otherwise, we say that this type A is* empty*.*

Conjecture 3 (Undecidability of top-sort inhabitation) *Deciding whether a top-sort is inhabited is undecidable.*

From now on, we will use the excluded middle to decide whether a top-sort is inhabited. However, in practice, this instance of the excluded middle can be removed safely because it is easy to decide whether a top-sort is inhabited.

Theorem 2.1.5 *A specification $\mathscr{C}$ is always equivalent to a specification $\mathscr{C}'$ where all the* empty *top-sorts have been removed.*

Proof *Given the derivation tree π of the $\Gamma \vdash_{\mathscr{C}} t : A$. By definition, A cannot be an* empty *top-sort, hence we can conclude.*

Definition 2.1.8 (Canonical inhabitant)
For every inhabited top-sort s in a specification $\mathscr{C}$, we can identify one canonical inhabitant. We denote $[s]^{\mathscr{C}}$ this canonical inhabitant. We denote $[s]^{\mathscr{C}}_{ctx}$ the typing context associated to this canonical inhabitant, hence we have: $[s]^{\mathscr{C}}_{ctx} \vdash_{\mathscr{C}} [s]^{\mathscr{C}} : s$.

The substitution of a variable by the canonical inhabitant of a top-sort might increase the typing context. This is why we have the following definitions.

Definition 2.1.9 (Substitution for canonical inhabitants)
We define $[\sigma]^{\mathscr{C}}$ a substitution which maps variables to canonical inhabitants of top-sorts. We define $\Gamma[\sigma]^{\mathscr{C}}$ the typing context substitution by induction on $[\sigma]^{\mathscr{C}}$:

- $\Gamma[\cdot]^{\mathscr{C}} := \Gamma$
- $\Gamma[\{X \leftarrow [s]^{\mathscr{C}}\}; \sigma]^{\mathscr{C}} := [s]^{\mathscr{C}}_{ctx} \sqcup \Gamma[\sigma]^{\mathscr{C}}$

We use a disjoint union to avoid any shadowing of a variable.

2.1.5 Weak CTS equivalence

Definition 2.1.10 (Weak judgment $\star$-embedding)
For any specification $\mathscr{C}$, we say that the judgment $\Gamma \vdash_\star t : A$ is $(\star, \mathscr{C})$-weakly embedded if there exist $\Gamma', t', A', [\sigma]^{\mathscr{C}}$ such that $\Gamma' \vdash_{\mathscr{C}} t' : A'$ is derivable and $\Gamma[\sigma]^{\mathscr{C}} =_\star \Gamma'$, $t\sigma =_\star t'$ and $A\sigma =_\star A'$ where $[\sigma]^{\mathscr{C}}$ is a substitution from $\mathcal{V}$ to $\mathcal{T}$.

Remark 13 *If $[\sigma]^{\mathscr{C}}$ is the empty substitution, the notion of weak judgment embedding coincides with judgment embedding. In particular weak judgment embeddings and judgment embeddings coincide when Γ is empty.*

Remark 14 *We use the notion of canonical inhabitant for a top-sort so that $[\sigma]^{\mathscr{C}}$ can be defined as a function. In the rest of this chapter, we will not insist on this point for sake of simplicity.*

Definition 2.1.11 (Weak judgment embedding)
For any CTS specification $\mathscr{C}$ and $\mathscr{C}$', a judgment $\Gamma \vdash_{\mathscr{C}} t : A$ is $(\mathscr{C}, \mathscr{C}')$-weakly embedded if the judgment $\Gamma_\star \vdash_\star t_\star : A_\star$ is $(\star, \mathscr{C}')$-weakly embedded.

Example 2.6 *Taking back specifications $\mathscr{D}_1$ and $\mathscr{D}_2$ presented in Example 2.3, the judgment $X : s_2 \vdash_{\mathscr{D}_1} X \to X : s_4$ is $(\mathscr{D}_1, \mathscr{D}_2)$-weakly embedded with the substitution $[\{X \leftarrow t_1\}]$. Indeed, the judgment $\vdash_{\mathscr{D}_2} t_1 \to t_1 : t_3$ is derivable.*

Definition 2.1.12 (Weak CTS embedding & Weak CTS equivalence)
For any two CTS specifications $\mathscr{C}$ and $\mathscr{C}$', we say that $\mathscr{C}$ is weakly embedded *into $\mathscr{C}$' if any derivable judgment $\Gamma \vdash_{\mathscr{C}} t : A$ is $(\mathscr{C}, \mathscr{C}')$-weakly embedded. Two CTS are said* weak CTS equivalent *if one is CTS embedded into the other and vice versa.*

Notation 15 *We denote $\mathscr{C} \trianglelefteq^w \mathscr{C}'$ if $\mathscr{C}$ is weakly CTS embedded into $\mathscr{C}'$. We denote $\mathscr{C} \sim^w \mathscr{C}'$ if $\mathscr{C}$ is weakly CTS equivalent to $\mathscr{C}'$.*

Our main interest for weak CTS equivalence is the Theorem 2.2.20.

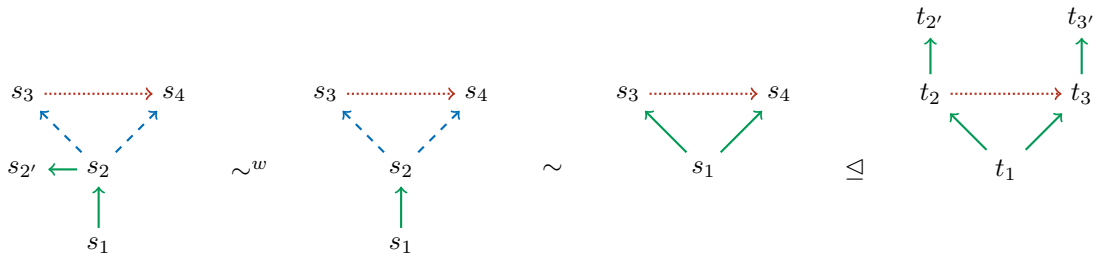


Figure 2.1: Proof that $\mathscr{D}_1 \trianglelefteq^w \mathscr{D}_2$

2.2 Meta-theory of equivalences

In this section we prove several results about equivalences we have defined previously.

Definition 2.2.1 (Strict weak CTS embedding)
The strict version of $\trianglelefteq^w$ (resp. $\trianglelefteq$) is defined as $\mathscr{C} \triangleleft^w \mathscr{C}' := \mathscr{C} \trianglelefteq^w \mathscr{C}' \wedge \neg(\mathscr{C}' \trianglelefteq^w \mathscr{C})$.

Theorem 2.2.1 *$\trianglelefteq^w$ is a preorder.*

Proof *Clearly, $\trianglelefteq^w$ is reflexive. Transitivity is a direct consequence of the definition of judgment embedding.*

Lemma 2.2.2 *If $\mathscr{C} \trianglelefteq_\sigma \mathscr{C}'$ then $\mathscr{C} \trianglelefteq \mathscr{C}'$. If $\mathscr{C} \trianglelefteq \mathscr{C}'$ then $\mathscr{C} \trianglelefteq^w \mathscr{C}'$.*

Proof *Direct consequence of the definition of $\trianglelefteq_\sigma$, $\trianglelefteq$ and $\trianglelefteq^w$.*

Example 2.7 *Using the theorems we will develop in the next part of this section, we are able to prove that $\mathscr{D}_1 \trianglelefteq^w \mathscr{D}_2$ as witnessed in Figure 2.1. The first equivalence says that one axiom is not necessary. This is possible because giving a type to a top-sort which is inhabited and which has no product associated with, does not increase the expressivity of the specification. As we hinted previously, s_1 can be substituted for any variable of type s_2. The second embedding is a particular case of functionalization of a specification. The second specification is the functional representation of the third specification which are equivalent. Since the third specification is included in the last one, we can deduce the existence of a specification morphism.*

In Figure 2.2, we give a proof that $\mathscr{D}_2 \trianglelefteq^w \mathscr{D}_1$ which uses the same ideas as previously.

A direct consequence of the definition of CTS embedding is the following theorem.

Theorem 2.2.3 (Preservation of termination) *If $\mathscr{C} \trianglelefteq^w \mathscr{C}'$ and $\mathscr{C}'$ is a terminating CTS, then so is $\mathscr{C}$.*

Proof *Suppose that there exist a term t such that $\Gamma \vdash_{\mathscr{C}} t : A$ but t is not SN. Then, by definition of weakly-embedding, we have that there exist $[\sigma]^{\mathscr{C}'}, \Gamma', t', A'$ such that $\Gamma' \vdash_{\mathscr{C}'} t' : A$ and $\Gamma[\sigma]^{\mathscr{C}'} =_\star \Gamma'$, $t\sigma =_\star t'$ and $A[\sigma]^{\mathscr{C}'} =_\star A'$. But if t is not SN, neither is $t\sigma$. We conclude with Lemma (2.1.3).*

Corollary 2.2.4 *If $\mathscr{C} \trianglelefteq \mathscr{C}'$ and $\mathscr{C}'$ is a terminating specification, then so is $\mathscr{C}$. If $\mathscr{C} \trianglelefteq_\sigma \mathscr{C}'$ and $\mathscr{C}'$ is a terminating specification, then so is $\mathscr{C}$.*

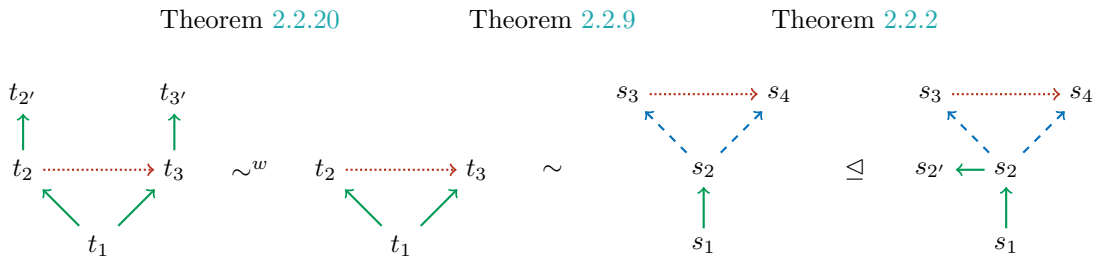


Figure 2.2: Proof that $\mathscr{D}_2 \trianglelefteq^w \mathscr{D}_1$

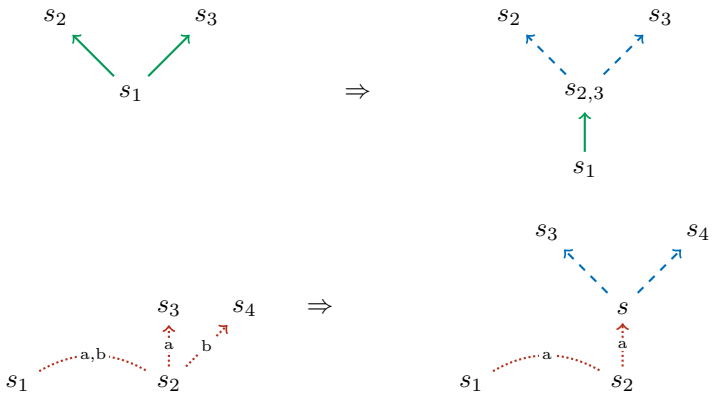


Figure 2.3: Idea behind functionalization of CTS specification

Remark 15 *For each notion of embedding $\trianglelefteq_\sigma$, $\trianglelefteq$ or $\trianglelefteq^w$, a category can be defined where:*

- *Objects are CTS specifications*
- *There is a morphism from $\mathscr{C}$ to $\mathscr{C}'$ if $\mathscr{C} \trianglelefteq_\sigma \mathscr{C}'$ (resp. $\trianglelefteq$, $\trianglelefteq^w$)*

and where $\mathscr{U}$ is a terminal object and the CTS where $\mathcal{S} = \emptyset$ is the initial object. We think that there is a deeper connection with category theory but we have not explored this path yet.

2.2.1 Functionalization

Our notion of CTS equivalence ($\trianglelefteq$) allows us to derive that any CTS specification is equivalent to a functional CTS where the relations $\mathcal{A}$ and $\mathcal{R}$ are functions (Definition 1.3.6). The intuition is that the non-functionality of the relations $\mathcal{A}$ and $\mathcal{R}$ can be transferred to the relation $\mathcal{C}$. This intuition is given in Figure 2.3: Every time we have $(s_1, s_2) \in \mathcal{A}$ and $(s_1, s_3) \in \mathcal{A}$, we add a new sort $s_{2,3}$. Then we replace these two axioms by the axiom $(s_1, s_{2,3})$, and we add $(s_{2,3}, s_3)$ and $(s_{2,3}, s_2)$ in $\mathcal{C}$. A similar transformation can be done for $\mathcal{R}$.

Formally, we define a function $\mathscr{F}$. which maps a specification to a functional specification.

Notation 16 *We use the following notations:*

$$\begin{aligned}
\mathcal{A}_{\mathscr{C}}^{s^+} &:= \{(s,s') \mid \forall s', (s,s') \in \mathcal{A}_{\mathscr{C}}\} \\
\mathcal{R}_{\mathscr{C}}^{(s,s')^+} &:= \{(s,s',s'') \mid \forall s'', (s,s',s'') \in \mathcal{R}_{\mathscr{C}}\} \\
\mathcal{S}_{\perp}^{\mathcal{A}} &:= \{\perp_s^{\mathcal{A}} \mid \forall s, |\mathcal{A}_{\mathscr{C}}^{s^+}| > 1\} \\
\mathcal{S}_{\perp}^{\mathcal{R}} &:= \{\perp_{(s,s')}^{\mathcal{R}} \mid \forall s, \forall s', |\mathcal{R}_{\mathscr{C}}^{(s,s')^+}| > 1\} \\
\mathcal{A}_{\mathscr{C}}^{\perp} &:= \{(s, \perp_s^{\mathcal{A}}) \mid \forall s, |\mathcal{A}_{\mathscr{C}}^{s^+}| > 1\} \\
\mathcal{R}_{\mathscr{C}}^{\perp} &:= \{(s,s', \perp_{(s,s')}^{\mathcal{R}}) \mid \forall s, \forall s', |\mathcal{R}_{\mathscr{C}}^{(s,s')^+}| > 1\} \\
\mathcal{C}_{\mathscr{C}}^{\perp^{\mathcal{A}}} &:= \{(\perp_s^{\mathcal{A}}, s') \mid \forall s, |\mathcal{A}_{\mathscr{C}}^{s^+}| > 1 \wedge (s,s') \in \mathcal{A}_{\mathscr{C}}^{s^+}\} \\
\mathcal{C}_{\mathscr{C}}^{\perp^{\mathcal{R}}} &:= \{(\perp_{(s,s')}^{\mathcal{R}}, s'') \mid \forall s, \forall s', |\mathcal{R}_{\mathscr{C}}^{(s,s')^+}| > 1 \wedge (s,s',s'') \in \mathcal{R}_{\mathscr{C}}^{(s,s')^+}\}
\end{aligned}$$

Definition 2.2.2 (CTS functionalization)
We define $\mathscr{F}_{\mathscr{C}}$ as:

$$\mathscr{F}_{\mathscr{C}} = \begin{cases} \mathcal{S} & = \mathcal{S}_{\mathscr{C}} \cup \mathcal{S}_{\perp}^{\mathcal{A}} \cup \mathcal{S}_{\perp}^{\mathcal{R}} \\ \mathcal{A} & = \left(\mathcal{A}_{\mathscr{C}} \setminus \bigcup_{|\mathcal{A}_{\mathscr{C}}^{s^+}|>1} \mathcal{A}_{\mathscr{C}}^{s}\right) \cup \mathcal{A}_{\mathscr{C}}^{\perp} \\ \mathcal{R} & = \left(\mathcal{R}_{\mathscr{C}} \setminus \bigcup_{|\mathcal{R}_{\mathscr{C}}^{(s,s')^+}|>1} \mathcal{R}_{\mathscr{C}}^{s,s'}\right) \cup \mathcal{R}_{\mathscr{C}}^{\perp} \\ \mathcal{C} & = \mathcal{C}_{\mathscr{C}} \cup \mathcal{C}_{\mathscr{C}}^{\perp^{\mathcal{A}}} \cup \mathcal{C}_{\mathscr{C}}^{\perp^{\mathcal{R}}} \end{cases}$$

Proving the left embedding $\mathscr{C} \trianglelefteq \mathscr{F}_{\mathscr{C}}$ is easy. The reason is because in this case, we don't need to change the sorts, only the derivation trees change.

Lemma 2.2.5 *If $\Gamma \vdash_{\mathscr{C}} t : A$ then $\Gamma \vdash_{\mathscr{F}_{\mathscr{C}}} t : A$.*

Proof *By induction on $\Gamma \vdash_{\mathscr{C}} t : A$. We give a proof only for the interesting rules: $\mathscr{C}_{sort}$,$\mathscr{C}_{\Pi}$.*

- ◊ *$\mathscr{C}_{sort}$: $t = s, A = s'$*

  By cases analysis on $(s,s') \in \mathcal{A}_{\mathscr{F}_{\mathscr{C}}}$.

  - □ *$(s,s') \in \mathcal{A}_{\mathscr{F}_{\mathscr{C}}}$:*
    We conclude with $\mathscr{C}_{sort}$.
  - □ *$(s,s') \notin \mathcal{A}_{\mathscr{F}_{\mathscr{C}}}$:*
    Then we have $|\mathcal{A}_{\mathscr{C}}^{s^+}| > 1$. Hence $(s, \perp_s^{\mathcal{A}}) \in \mathcal{A}_{\mathscr{F}_{\mathscr{C}}}$ and $(\perp_s^{\mathcal{A}}, s') \in \mathcal{C}\mathscr{F}_{\mathscr{C}}$. Hence we can conclude with $\mathscr{C}_{sort}$ and $\mathscr{C}_{\trianglelefteq}^{s}$.

- ◊ *$\mathscr{C}_{\Pi}$: $t = (x:A) \to B, A = s$*

  By case analysis on $(s,s',s'') \in \mathcal{R}_{\mathscr{F}_{\mathscr{C}}}$. The proof is similar to the case $\mathscr{C}_{sort}$.

Lemma 2.2.6 *The judgment $\Gamma \vdash_{\mathscr{C}} t : A$ is $(\mathscr{C}, \mathscr{F}_{\mathscr{C}})$ – embeddable.*

Proof *Direct consequence of Lemma 2.2.5.*

For the right embedding $\mathscr{F}_{\mathscr{C}} \trianglelefteq \mathscr{C}$, the proof is a bit more difficult because we added new sorts in $\mathscr{F}_{\mathscr{C}}$ which, a priori, could be used to derive new judgments. However, because these sorts are top-sorts, they can appear only on the right-hand side of a judgment. Hence, we can prove that these sorts can always be replaced by one of its direct successors in the cumulativity relation $\mathcal{C}_{\mathscr{F}_{\mathscr{C}}}$.

Lemma 2.2.7 *If $\Gamma \vdash_{\mathscr{F}_{\mathscr{C}}} t : A$ then $\Gamma \vdash_{\mathscr{C}} t : A'$ where A' is either:*

- *s if $A \in \mathcal{S}_\perp^{\mathcal{A}}$ with $(A, s) \in \mathcal{C}_{\mathscr{C}}^{\perp^{\mathcal{A}}}$*
- *s if $A \in \mathcal{S}_\perp^{\mathcal{R}}$ with $(A, s) \in \mathcal{C}_{\mathscr{C}}^{\perp^{\mathcal{R}}}$*
- *A otherwise*

Proof *By induction on $\Gamma \vdash_{\mathscr{F}_{\mathscr{C}}} t : A$. The proof is similar to Lemma (2.2.5), except for the rule $\mathscr{C}_\Pi$ which is interesting.*

◊ *$\mathscr{C}_\Pi$: $t = (x : B) \to C, A = s'$*

Then $\Gamma \vdash_{\mathscr{C}} A : s_1$ and $\Gamma, x : A \vdash_{\mathscr{C}} B : s_2$ and $(s_1, s_2, s') \in \mathcal{R}_{\mathscr{F}_{\mathscr{C}}}$. By definition of $\mathscr{F}_{\mathscr{C}}$, $s_1, s_2 \notin \mathcal{S}_\perp^{\mathcal{A}} \cup \mathcal{S}_\perp^{\mathcal{R}}$. By case analysis on $s' \in \mathcal{S}_\perp^{\mathcal{R}}$.

- □ *$s' \in \mathcal{S}_\perp^{\mathcal{R}}$:*
  For any s such that $(s', s) \in \mathcal{C}_{\mathscr{F}_{\mathscr{C}}}$, we have $(s_1, s_2, s) \in \mathcal{R}_{\mathscr{C}}$. Hence we can conclude with $\mathscr{C}_\Pi$.
- □ *$s' \notin \mathcal{S}_\perp^{\mathcal{R}}$:*
  Hence $(s_1, s_2, s') \in \mathcal{R}_{\mathscr{C}}$. We can conclude with $\mathscr{C}_\Pi$.

Lemma 2.2.8 *The judgment $\Gamma \vdash_{\mathscr{F}_{\mathscr{C}}} t : A$ is $(\mathscr{F}_{\mathscr{C}}, \mathscr{C})-$ embeddable.*

Proof *Direct consequence of Lemma (2.2.7).*

Theorem 2.2.9 *$\mathscr{C} \sim \mathcal{F}_{\mathscr{C}}$*

Proof *Consequence of Lemma (2.2.6) and Lemma (2.2.8).*

Theorem 2.2.10 *If $\mathscr{C}$ is decidable specification, so is $\mathcal{F}_{\mathscr{C}}$.*

Proof *One can check that all the cases from decidable CTS specification are easily checked.*

2.2.2 Injectivization

Injectivity for CTS is defined by Barthe in [Bar99b] where it requires that the relation $\mathcal{R}$ is injective on the second argument. Here, we show that we can do a similar process to create for every CTS specification, an equivalent one which is injective (for products, it is injective on the second argument). The idea of this process is described in Fig. 2.4. The formal definition is given below:

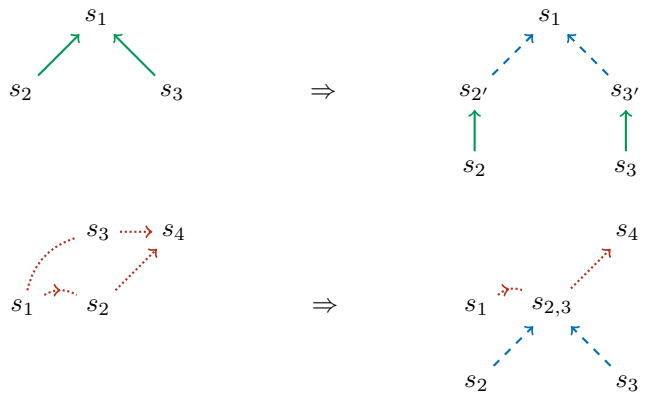


Figure 2.4: Idea behind injectivization of CTS specification

Notation 17 *We define the following notations*

$$
\begin{aligned}
\mathcal{A}_{\mathscr{C}}^{s'^{-}} &:= \{(s,s') \mid \forall s,(s,s') \in \mathcal{A}_{\mathscr{C}}\} \\
\mathcal{R}_{\mathscr{C}}^{(s,s'')^{-}} &:= \{(s,s',s'') \mid \forall s',(s,s',s'') \in \mathcal{R}_{\mathscr{C}}\} \\
\mathcal{S}_{\top}^{\mathcal{A}} &:= \{\top_s^{\mathcal{A}} \mid \forall s',(s,s') \in |\mathcal{A}_{\mathscr{C}}^{s'^{-}}| > 1\} \\
\mathcal{S}_{\top}^{\mathcal{R}} &:= \{\top_{(s,s'')}^{\mathcal{R}} \mid \forall s,\forall s'',|\mathcal{R}_{\mathscr{C}}^{(s,s'')^{-}}| > 1\} \\
\mathcal{A}_{\mathscr{C}}^{\top} &:= \{(s,\top_s^{\mathcal{A}}) \mid \forall s,\forall s',(s,s') \in \mathcal{A}_{\mathscr{C}}^{s'^{-}} \wedge |\mathcal{A}_{\mathscr{C}}^{s'^{-}}| > 1\} \\
\mathcal{R}_{\mathscr{C}}^{\top} &:= \{(s,\top_{(s,s'')}^{\mathcal{R}},s'') \mid \forall s,\forall s'',|\mathcal{R}_{\mathscr{C}}^{(s,s'')^{-}}| > 1\} \\
\mathcal{C}_{\mathscr{C}}^{\top^{\mathcal{A}}} &:= \{(\top_s^{\mathcal{A}},s') \mid \forall s,\forall s',|\mathcal{A}_{\mathscr{C}}^{s'^{-}}| > 1 \wedge (s,s') \in \mathcal{A}_{\mathscr{C}}^{s^{-}}\} \\
\mathcal{C}_{\mathscr{C}}^{\top^{\mathcal{R}}} &:= \{(s',\top_{(s,s'')}^{\mathcal{R}}) \mid \forall s,\forall s',\forall s'',|\mathcal{R}_{\mathscr{C}}^{(s,s'')^{-}}| > 1 \wedge (s,s',s'') \in \mathcal{R}_{\mathscr{C}}^{(s,s'')^{-}}\}
\end{aligned}
$$

Definition 2.2.3 (CTSinjectivization)
We define $\mathscr{I}_{\mathscr{C}}$ as:

$$
\mathscr{I}_{\mathscr{C}} = \begin{cases}
\mathcal{S} & = \mathcal{S}_{\mathscr{C}} \cup \mathcal{S}_{\top}^{\mathcal{A}} \cup \mathcal{S}_{\top}^{\mathcal{R}} \\
\mathcal{A} & = \left(\mathcal{A}_{\mathscr{C}} \setminus \bigcup_{|\mathcal{A}_{\mathscr{C}}^{s'^{-}}|>1} \mathcal{A}_{\mathscr{C}}^{s'^{-}}\right) \cup \mathcal{A}_{\mathscr{C}}^{\top} \\
\mathcal{R} & = \left(\mathcal{R}_{\mathscr{C}} \setminus \bigcup_{|\mathcal{R}_{\mathscr{C}}^{(s,s'')^{-}}|>1} \mathcal{R}_{\mathscr{C}}^{(s,s'')^{-}}\right) \cup \mathcal{R}_{\mathscr{C}}^{\top} \\
\mathcal{C} & = \mathcal{C}_{\mathscr{C}} \cup \mathcal{C}_{\mathscr{C}}^{\top^{\mathcal{A}}} \cup \mathcal{C}_{\mathscr{C}}^{\top^{\mathcal{R}}}
\end{cases}
$$

Theorem 2.2.11 $\mathscr{C} \sim \mathcal{I}_{\mathscr{C}}$

Proof *The proof is similar to 2.2.9.*

Theorem 2.2.12 *If $\mathscr{C}$ is decidable, so is $\mathscr{I}_{\mathscr{C}}$.*

Proof *Routine check.*

Theorem 2.2.13 *If $\mathscr{C}$ is functional so is $\mathscr{I}_{\mathscr{C}}$.*

Proof *Routine check.*

Theorem 2.2.14 *For every CTS specification $\mathscr{C}$, there exist a CTS specification $\mathscr{C}'$ which is functional and injective such that $\mathscr{C} \sim \mathscr{C}'$.*

Proof $\mathscr{C}' = \mathscr{I}_{\mathscr{F}_{\mathscr{C}}}$.

Top-sorts and cumulativity

The purpose of this section is to address the case where a top-sort is a subtype of a sort which has a type as in the left part of Fig. 2.5. When we defined functionalization and injectivization, that happened every time we introduced a new sort. This situation raises an issue for bi-directional CTS defined in Chapter 4. The idea of bi-directional CTS is to push back the use of cumulativity only at the end of type checking or on the right-hand side of an application. However, this is not possible with the situation above: One may need to use cumulativity before using an axiom. Hence, the idea is to have the transformation presented in Fig. 2.5.

This transformation is not a CTS equivalence ($\trianglelefteq$) because typing a top-sort allows to see this top-sort as a type itself. Hence, we may introduce variables which have this *top-sort* as a type which is not possible before (this problem is raised in Example 2.5). This is formalized in the example below.

Example 2.8 *In the specification* **HOL**, *one can derive the judgment $X : \Box \vdash_{\mathbf{HOL}} X \to X : \Box$. Such judgment is not derivable in the specification ω because $\Box$ is a top-sort in ω. However, for every term t such that $\Gamma \vdash_{\omega} t : \Box$, we can derive $\Gamma \vdash_{\omega} t \to t : \Box$.*

Theorem 2.2.15 *Any CTS where all top-sorts are inhabited is weakly equivalent to a CTS where all top-sorts are inhabited by a sort.*

Proof *Given a top-sort s which is not inhabited by a sort. We add a sort s' and the axiom (s', s). We repeat this construction for all (possible infinite) top-sorts of the initial specification. We call this new specification $\mathscr{C}'$. By definition, there is a sort-morphism from $\mathscr{C}$ to $\mathscr{C}'$ which is an inclusion. Now, we prove that $\mathscr{C}'$ is weakly CTS embedded to $\mathscr{C}$. Given a judgment $\Gamma \vdash_{\mathscr{C}'} t : A$, every time a sort $s' \notin \mathcal{S}_{\mathscr{C}}$ appears, then there exists s in $\mathcal{S}_{\mathscr{C}}$ such that $(s', s) \in \mathcal{A}_{\mathscr{C}'}$. We replace this sort by the canonical inhabitant of s (which exists since the top-sort is inhabited). We obtain a new judgment which is well-typed in $\mathscr{C}$.*

Definition 2.2.4 (Top-sort regular CTS)
We say that a specification $\mathscr{C}$ is top-sort regular *if all the top-sorts of $\mathscr{C}$ are inhabited by a sort.*

Corollary 2.2.16 *From Theorem 2.1.5 and Theorem 2.2.15, every CTS specification is weakly CTS equivalent to a top-sort regular CTS.*

The transformation induced by Fig. 2.5 can only imply a weak equivalence.

Figure 2.5: Top-sort and cumulativity

Notation 18 *We define the following notations*

$$\mathcal{S}^{\mathcal{S}^{\top}_{\mathscr{C}}} := \{\top_s \mid \forall s, s \in \mathcal{S}^{\top}_{\mathscr{C}}\}$$
$$\mathcal{A}^{\mathcal{S}^{\top}_{\mathscr{C}}} := \{(s, \top_s) \mid \forall s, s \in \mathcal{S}^{\top}_{\mathscr{C}}\}$$
$$\mathcal{C}^{\mathcal{S}^{\top}_{\mathscr{C}}} := \{(\top_s, s'') \mid \forall s, \forall s', \forall s'', s \in \mathcal{S}^{\top}_{\mathscr{C}}, (s, s') \in \mathcal{C}, (s', s'') \in \mathcal{A}\}$$

Definition 2.2.5 (CTS with at most one top-sort)
We define $\mathscr{T}_{\mathscr{C}}$ as:

$$\mathscr{T}_{\mathscr{C}} = \begin{cases} \mathcal{S} & = \mathcal{S}_{\mathscr{C}} \cup \mathcal{S}^{\mathcal{S}^{\top}_{\mathscr{C}}} \\ \mathcal{A} & = \mathcal{A}_{\mathscr{C}} \cup \mathcal{A}^{\mathcal{S}^{\top}_{\mathscr{C}}} \\ \mathcal{R} & = \mathcal{R}_{\mathscr{C}} \\ \mathcal{C} & = \mathcal{C}_{\mathscr{C}} \cup \mathcal{C}^{\mathcal{S}^{\top}_{\mathscr{C}}} \end{cases}$$

Lemma 2.2.17 $\mathscr{C} \trianglelefteq \mathscr{T}_{\mathscr{C}}$

Proof *Trivial since we have a direct inclusion from $\mathscr{C}$ to $\mathscr{T}_{\mathscr{C}}$.*

Lemma 2.2.18 *If $\mathscr{C}$ is* top-sort regular *then $\mathscr{T}_{\mathscr{C}} \trianglelefteq^{w} \mathscr{C}$*

Proof *Sketch of proof. Every time we have a variable $x : s'$ with $s' \in \mathcal{S}^{\mathcal{S}^{\top}_{\mathscr{C}}}$, there exists a derivation $\vdash_{\mathscr{C}} t : s$ since $\mathscr{C}$ is* top-sort regular*. Hence, we can substitute t for this variable.*

Theorem 2.2.19 *If $\mathscr{C}$ is* top-sort regular *then $\mathscr{T}_{\mathscr{C}} \sim^{w} \mathscr{C}$*

Proof *Direct consequence of Lemma 2.2.17 and 2.2.18.*

One can reuse the results above for top-sorts without taking cumulativity into account. This gives us the following result:

Theorem 2.2.20 *For any* top-sort regular *CTS $\mathscr{C}$, for any $s' \in \mathcal{S}$ such that $(s, s') \in \mathcal{A}$, we can define the specification $\mathscr{C}'$ with a new sort s'' such that $(s', s'') \in \mathcal{A}$. Then $\mathscr{C} \sim^{w} \mathscr{C}'$.*

Proof *One can reuse the proof of Lemma 2.2.18.*

Corollary 2.2.21 *Any* top-sort regular *CTS is weakly equivalent to a CTS with at most one top-sort.*

Proof *The construction adds a new sorts s_∞ and axioms (s, s_∞) whenever $s \in \mathcal{S}^\top_{\mathscr{C}}$. The proof is a consequence of Theorem 2.2.20.*

Theorem 2.2.22 *Any* top-sort regular *CTS is weakly equivalent to a CTS without top-sort.*

Proof *Sketch of proof. It is sufficient to iterate the construction presented in Theorem 2.2.21.*

Remark 16 *If we are interested only in judgments in a closed typing context, one can drop the condition that the specification is* top-sort regular *and judgments are equivalent.*

2.3 Deciding judgment embeddings

Interoperability between CTS specifications requires to decide whether a typing judgment can be $(\mathscr{C}, \mathscr{C}')$-embedded[2]. However, from our definition of $(\mathscr{C}, \mathscr{C}')$-embedded, it is complicated to know whether a judgment is embeddable without having a derivation of this judgment. In this section, we assume the existence of a type checker, meaning a procedure that returns a derivation of a judgment whenever such derivation exists. Our method relies on this derivation tree to decide whether this tree can be $(\mathscr{C}, \mathscr{C}')$-embedded. If the answer is positive, then we can conclude that the judgment itself can be $(\mathscr{C}, \mathscr{C}')$-embedded. However, if the answer is negative, it might be because we have picked the wrong derivation tree. A discussion about choosing the correct derivation tree is made in Section 2.4.

The main idea behind our method is to derive a *free CTS* specification from a derivation tree. Then we show that deciding if a judgment is $(\mathscr{C}, \mathscr{C}')$-embedded is the same as finding a specification morphism from this free CTS to $\mathscr{C}'$. Our method needs to compute the normal form of a term, for this reason we assume that the specification is SN. This restriction is not a problem in practice since all the specifications implemented by concrete systems are SN.

The idea behind the *free CTS* of a derivation tree $\mathcal{T}$ is to replace every occurrence of a sort by a fresh sort getting thus also a new judgment. The free CTS is the specification that makes this new judgment derivable by completing the axioms, the rules and the cumulativity relation with what is needed. Modulo the name of the sorts, this specification is unique. However, the free CTS of a derivation tree is not easy to define. For example, assume we have a derivation of $f : \star \to \star, x : \star \vdash_{\mathscr{C}} f\ x : \star$. Generating fresh variables first for the derivation of f leads to the following judgment $f : ?_1 \to ?_2, x :?_3 \vdash_{\mathscr{C}} f : ?_1 \to ?_2$ and doing the same thing for x leads to this other judgment $f : ?_4 \to ?_5, x :?_6 \vdash_{\mathscr{C}} x :?_6$. Because a priori $?_3 \neq ?_6$, the term $f\ x$ is not well-typed anymore. To solve this issue, we have to generate an equivalence relation between sorts and use it to quotient the sorts of the free CTS.

Definition 2.3.1 (Free equivalence relation for conversion)
Given two terms t and t' such that $t =_\star t'$, we define recursively an equivalence relation on sorts denoted $\equiv_{t,t'}$ as follows:

$$
\begin{aligned}
\equiv_{s,s'} &:= && (s,s') \cup (s',s) \cup (s,s) \cup (s',s') \\
\equiv_{x,y} &:= && \emptyset \\
\equiv_{(x:A)\to B,(y:A')\to B'} &:= && \equiv_{A,A'} \cup \equiv_{B,B'} \\
\equiv_{\lambda x:A.\,t,\lambda y:A'.\,t'} &:= && \equiv_{A,A'} \cup \equiv_{t,t'} \\
\equiv_{t\ u,t'\ u'} &:= && \equiv_{t,t'} \cup \equiv_{u,u'}
\end{aligned}
$$

[2]We will not consider the weak version in this section

This definition is well-formed because $=_\star$ is stable by sub-term and all the other cases are not possible. This definition is extended in a natural way to typing contexts.

Lemma 2.3.1 *If $t =_\star t'$ then $\equiv_{t,t'}$ is an equivalence relation on sorts.*

Proof *By induction on t. All the cases are trivial.*

Notation 19 *If $\mathscr{C}$ is a CTS specification and $\equiv$ is an equivalence relation on sorts, we denote $\mathscr{C}/_\equiv$ the new CTS specification such that $\mathcal{S}_{\left(\mathscr{C}/_\equiv\right)} := (\mathcal{S}_\mathscr{C})/_\equiv$ and equal otherwise.*

We also need to generate equalities when $A \preceq_\mathscr{C} B$. This is where we use our assumption that $\mathscr{C}$ is SN. The fact that $\mathscr{C}$ is SN gives us a canonical way to compute these equalities by using their normal form[3].

Definition 2.3.2 (Free equivalence relation and free cumulativity relation for subtyping)

Let have A and B such that $A \preceq_\mathscr{C} B$ and we denote $A\downarrow$ and $B\downarrow$ their normal form. We define $\equiv^{\preceq_\mathscr{C}}_{A,B}$ by induction on their normal form as follows:

$$\begin{array}{rll} \equiv^{\preceq_\mathscr{C}}_{s,s'} := \emptyset & & \text{when } A\downarrow = s \text{ and } B\downarrow = s' \\ \equiv^{\preceq_\mathscr{C}}_{(x:C)\to D,(x:E)\to F} := \equiv_{C,E} \cup \equiv^{\preceq_\mathscr{C}}_{D,F} & & \text{when } A\downarrow = (x:C)\to D \text{ and } B\downarrow = (x:E)\to F \\ \equiv^{\preceq_\mathscr{C}}_{A,B} := \equiv_{A\downarrow,B\downarrow} & & \text{otherwise} \end{array}$$

We define $\preceq_\mathscr{C} A,B$ by induction on the normal form of A and B:

$$\begin{array}{rll} \preceq_\mathscr{C} s,s' := (s,s') & & \text{when } A\downarrow = s \text{ and } B\downarrow = s' \\ \preceq_\mathscr{C} (x:C)\to D,(x:E)\to F := \preceq_\mathscr{C} D,F & & \text{when } A\downarrow = (x:C)\to D \text{ and } B\downarrow = (x:E)\to F \\ \preceq_\mathscr{C} A,B := \emptyset & & \text{otherwise} \end{array}$$

The function $\equiv^{\preceq_\mathscr{C}}_{A,B}$ intends to capture all the equalities necessary so that in the free CTS, $A' \equiv_\beta B'$ if $A \equiv_\beta B$. However, when $A \preceq_\mathscr{C} B$ but $A \not\equiv_\beta B$, then we want to equate every sorts except the *last ones*. This is why we add these *fresh sorts* in the cumulativity relation in the free CTS.

Notation 20 *If $\mathscr{C}$ and $\mathscr{C}'$ are two specifications and X is a set we use the following notations:*

- *$\mathscr{C} \cup \mathscr{C}'$ denotes their disjoint pairwise union*
- *$\mathscr{C} \cup_\mathcal{S} X$ denotes the specification $\mathscr{C}'$ which is equal to $\mathscr{C}$ except that $\mathcal{S}_{\mathscr{C}'} := \mathcal{S}_\mathscr{C} \cup X$*
- *$\mathscr{C} \cup_\mathcal{A} X$ is the same for $\mathcal{A}$*
- *$\mathscr{C} \cup_\mathcal{R} X$ is the same for $\mathcal{R}$*
- *$\mathscr{C} \cup_\mathcal{C} X$ is the same for $\mathcal{C}$*

[3] Maybe we could relax this hypothesis so that $\mathscr{C}$ is WN only

- *We denote the empty CTS by* $\emptyset$.

Defining the free CTS requires also to remember how the judgment was changed by introducing fresh variables. This is why our definition of free CTS returns the new specification as well as a new judgment which only makes sense in this specification.

Definition 2.3.3 (Free CTS)
Given a derivation tree $\mathcal{T}$ *of a judgment* $\Gamma \vdash_{\mathscr{C}} t : A$ *or a judgment* $\Gamma \vdash_{\mathscr{C}} \mathbf{wf}$, *we define recursively the free CTS denoted* $\mathscr{F}_{\mathcal{T}}$ *in Fig. 2.6. We denote* $\Gamma' \vdash_{\mathscr{F}_{\mathcal{T}}} t' : A'$ *the judgment computed by this function in the new specification* $\mathscr{F}_{\mathcal{T}}$.

Remark 17 *The way CTS are formulated with a judgment* $\Gamma \vdash_{\mathscr{C}} \mathbf{wf}$ *implies that when* $x : A \in \Gamma$ *there might be many derivations that* A *is well-sorted. Hence, the free CTS will generate many sorts for the type of* A *because of these verifications. It would not be the case if CTS were formulated with a weakening rule instead. This problem arises in theory, but does not appear in practice because type checkers are implemented in a way that it is checked only once that* A *is well-sorted. In our examples below, we will simplify the free CTS to avoid these redundancies.*

Remark 18 *The rules* $\mathscr{C}_{\preceq}$ *and* $\mathscr{C}^{s}_{\preceq}$ *requires to compute the normal form of* A *and* B. *It is possible because we assumed* $\mathscr{C}$ *was SN, however this does not scale up. We will come back to this problem in Chapter 10 when we describe our implementation.*

Remark 19 *One may wonder why we insisted by picking the normal form in the definition of free CTS for the rules* $\mathscr{C}_{\preceq}$ *and* $\mathscr{C}^{s}_{\preceq}$ *and not any term* C *such that* $A_\star \hookrightarrow^*_\beta C \hookleftarrow^*_\beta B_\star$. *There are several reasons: First, it gives us a unique way to define the free CTS. Second, it avoid problems where we could equate more things than necessary. For example if we pick* $A = (\lambda x : ?_2.\, y)\ ?_1$ *and* $B = (\lambda x : ?_3.\, y)\ ?_4$, *we could have picked* $C = (\lambda x : \star.\, y)\ \star$. *Hence, we need to add the equalities* $?_2 =?_3$ *and* $?_1 =?_4$. *While if we do a* β*-reduction first, we only need to add the equality* $?_2 =?_3$. *We think that the definition of free CTS could be extended for specifications which are not SN. However, it is not clear how to choose the equalities that should be generated.*

Lemma 2.3.2 *Given a derivation tree* $\mathcal{T}$ *of* $\Gamma \vdash_{\mathscr{C}} t : A$, *let* $\Gamma' \vdash_{\mathscr{F}_{\mathcal{T}}} t' : A'$ *its free CTS. Then this judgment is indeed derivable and* $\Gamma =_\star \Gamma'$, $t =_\star t'$ *and* $A =_\star A'$.

Proof *By induction on* $\mathcal{T}$.

Theorem 2.3.3 *A judgment* $\Gamma \vdash_{\mathscr{C}} t : A$ *is* $(\mathscr{C}, \mathscr{C}')$*-embeddable if it is derivable with a derivation tree* $\mathcal{T}$ *and if there is a specification morphism from* $\mathscr{F}_{\mathcal{T}}$ *to* $\mathscr{C}'$.

Proof *A direct consequence of Lemma (2.3.2).*

In the example below, we show why using a free CTS is interesting to decide if a judgment can be $(\mathscr{C}, \mathscr{C}')$-embedded.

Example 2.9 *We reuse the specifications* $\mathscr{D}_1$ *and* $\mathscr{D}_2$ *defined in Example 2.3. In* $\mathscr{D}_1$, *one can derive the judgment* $x : s_2, y : s_2 \vdash_{\mathscr{D}_1} x \to y : s_4$ *in this way:*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(s_2, s_2') \in \mathcal{A}_{\mathscr{D}_1}}}{\vdots}}{x : s_2, y : s_2 \vdash_{\mathscr{D}_2} x : s_2} \quad \dfrac{}{(s_2, s_3) \in \mathcal{C}_{\mathscr{D}_1}}}{x : s_2, y : s_2 \vdash_{\mathscr{D}_2} x : s_3} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{(s_2, s_2') \in \mathcal{A}_{\mathscr{D}_1}}}{\vdots}}{x : s_2, y : s_2, \cdot : s_3 \vdash_{\mathscr{D}_2} y : s_2} \quad \dfrac{}{(s_2, s_4) \in \mathcal{C}_{\mathscr{D}_1}}}{x : s_2, y : s_2, \cdot : s_3 \vdash_{\mathscr{D}_2} y : s_4} \quad \dfrac{}{(s_3, s_4, s_4) \in \mathcal{R}_{\mathscr{D}_1}}}{x : s_2, y : s_2 \vdash_{\mathscr{D}_1} x \to y : s_4}$$

$$\left[\!\!\left[\frac{}{\varnothing \vdash_{\mathscr{C}} \mathbf{wf}}\ \mathscr{C}^{\mathbf{wf}}_{\varnothing} \right]\!\!\right] := \varnothing \vdash_{\emptyset} \mathbf{wf}$$

$$\left[\!\!\left[\frac{\dfrac{\Pi_A}{\Gamma \vdash_{\mathscr{C}} A : s} \quad x \notin \Gamma}{\Gamma, x : A \vdash_{\mathscr{C}} \mathbf{wf}}\ \mathscr{C}^{\mathbf{wf}}_{var} \right]\!\!\right] := \Gamma', x : A' \vdash_{\mathcal{F}} \mathbf{wf}$$

where $\Gamma' \vdash_{\mathcal{F}} A' :?_i = [\Pi_A]$

$$\left[\frac{\dfrac{\Pi_\Gamma}{\Gamma \vdash_{\mathscr{C}} \mathbf{wf}} \quad (s_1, s_2) \in \mathcal{A}}{\Gamma \vdash_{\mathscr{C}} s_1 : s_2}\ \mathscr{C}_{sort} \right] := \Gamma' \vdash_{\mathcal{F}} ?_i :?_k$$

where $\Gamma' \vdash_{\mathcal{F}'} \mathbf{wf} = [\![\Pi_\Gamma]\!]$
and $\mathcal{F} = \mathcal{F}' \cup_{\mathcal{S}} \{?_i, ?_k\} \cup_{\mathcal{A}} \{(?_i, ?_k)\}$

$$\left[\frac{\dfrac{\Pi_A}{\Gamma \vdash_{\mathscr{C}} A : s_1} \quad \dfrac{\Pi_B}{\Gamma, x : A \vdash_{\mathscr{C}} B : s_2} \quad (s_1, s_2, s_3) \in \mathcal{R}}{\Gamma \vdash_{\mathscr{C}} (x : A) \to B : s_3}\ \mathscr{C}_{\Pi} \right] := \Gamma' \vdash_{\mathcal{F}} (x : A') \to B' :?_k$$

where $\Gamma' \vdash_{\mathcal{F}_A} A' :?_i = [\Pi_A]$
and $\Gamma'', x : A'' \vdash_{\mathcal{F}_B} B' :?_j = [\Pi_B]$
and $\mathcal{F} = (\mathcal{F}_A \cup \mathcal{F}_B \cup_{\mathcal{S}} \{?_k\} \cup_{\mathcal{R}} (?_i, ?_j, ?_k))/(\equiv_{A',A''} \cup \equiv_{\Gamma',\Gamma''})$

$$\left[\frac{\dfrac{\Pi_M}{\Gamma, x : A \vdash_{\mathscr{C}} M : B} \quad \dfrac{\Pi_{(x:A)\to B}}{\Gamma \vdash_{\mathscr{C}} (x : A) \to B : s}\ \mathscr{C}_{\lambda} \right] := \Gamma' \vdash_{\mathcal{F}} \lambda x : A'.\, u' : (x : A') \to B'$$

where $\Gamma', x : A' \vdash_{\mathcal{F}_M} u' : B' = [\Pi_M]$
and $\Gamma'' \vdash_{\mathcal{F}_M} (x : A'') \to B'' :?_i = [\Pi_{(x:A)\to B}]$
and $\mathcal{F} = (\mathcal{F}_M \cup \mathcal{F}_{(x:A)\to B})/(\equiv_{\Gamma',\Gamma''} \cup \equiv_{A',A''} \cup \equiv_{B',B''})$

$$\left[\frac{\dfrac{\Pi_M}{\Gamma \vdash_{\mathscr{C}} M : (x : A) \to B} \quad \dfrac{\Pi_N}{\Gamma \vdash_{\mathscr{C}} N : A}\ \mathscr{C}_{app} \right] := \Gamma' \vdash_{\mathcal{F}} M'\ N' :?_i$$

where $M \vdash_{\mathcal{F}_M} \Gamma' : (x : A') \to B' = [\Pi_M]$
and $\Gamma'' \vdash_{\mathcal{F}_N} N' : A'' = [\Pi_N]$
and $\mathcal{F} = (\mathcal{F}_M \cup \mathcal{F}_N)/(\equiv_{\Gamma',\Gamma''} \cup \equiv_{A,A'})$

$$\left[\frac{\dfrac{\Pi_M}{\Gamma \vdash_{\mathscr{C}} M : A} \quad \dfrac{\Pi_B}{\Gamma \vdash_{\mathscr{C}} B : s} \quad A \preceq_{\mathscr{C}} B}{\Gamma \vdash_{\mathscr{C}} M : B}\ \mathscr{C}_{\preceq} \right] := \Gamma' \vdash_{\mathcal{F}} M' : B'$$

where $\Gamma' \vdash_{\mathcal{F}_M} M' : A' = [\Pi_M]$
and $\Gamma'' \vdash_{\mathcal{F}_B} B' :?_i = [\Pi_B]$
and $\mathcal{F} = (\mathcal{F}_M \cup \mathcal{F}_B \cup_{\mathcal{C}} \preceq_{\mathscr{C}} A',B')/(\equiv_{\Gamma',\Gamma''} \cup \equiv^{\preceq_{\mathscr{C}}}_{A',B'})$

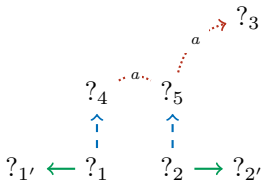
$$\left[\frac{\dfrac{\Pi_M}{\Gamma \vdash_{\mathscr{C}} M : A} \quad A \preceq_{\mathscr{C}} s}{\Gamma \vdash_{\mathscr{C}} M : s}\ \mathscr{C}^{s}_{\preceq} \right] := \Gamma' \vdash_{\mathcal{F}} M' :?_i$$

where $\Gamma' \vdash_{\mathcal{F}_M} M' : A' = [\Pi_M]$
and $\mathcal{F} = \mathcal{F}_M \cup_{\mathcal{S}} ?_i \cup_{\mathcal{C}} \preceq_{\mathscr{C}} A',?_i$

Figure 2.6: Free CTS

One can check that the free CTS obtained from this derivation allows to derive the judgment $x : ?_1, y : ?_2 \vdash_{\mathcal{F}_\mathcal{T}} x \to y : ?_3$ *where* $\mathcal{F}_\mathcal{T}$ *is defined as:*

$$(\mathcal{F}_\mathcal{T}) = \begin{cases} \mathcal{S} = & \{?_1, ?_{1'}, ?_2, ?_{2'}, ?_3, ?_4, ?_5\} \\ \mathcal{A} = & \{(?_1, ?_{1'}), (?_2, ?_{2'})\} \\ \mathcal{R} = & \{(?_4, ?_5, ?_3)\} \\ \mathcal{C} = & \{(?_1, ?_4), (?_2, ?_5)\} \end{cases}$$



The following function is a specification morphism from $\mathcal{F}_\mathcal{T}$ *to* $\mathscr{D}_2$*:*

$$\begin{array}{lll} ?_1 \to t_2 & & ?_2 \to t_3 \\ ?_{1'} \to t_{2'} & & ?_{2'} \to t_{3'} \\ ?_3 \to t_3 & ?_4 \to t_2 & ?_5 \to t_3 \end{array}$$

Hence the judgment $x : s_2, y : s_2 \vdash_{\mathscr{D}_1} x \to y : s_4$ *is* $(\mathscr{D}_1, \mathscr{D}_2)$*-embeddable.*

To decide whether a judgment is $(\mathscr{C}, \mathscr{C}')$-decidable is thus the same as deciding whether there is a specification morphism from this free CTS to $\mathscr{C}$'.

Theorem 2.3.4 *Assuming that* $\mathscr{C}'$ *is a decidable and finite specification then deciding whether a derivation tree* $\mathcal{T}$ *can be* $(\mathscr{C}, \mathscr{C}')$*-embedded is decidable.*

Proof *Since the free CTS is a finite specification, one can enumerate the functions from* $\mathcal{S}_{\mathscr{F}_\mathcal{T}}$ *to* $\mathscr{C}$*' and check whether this function is a specification morphism.*

In practice, $\mathscr{C}'$ is not always finite as $\mathscr{C}^C_\infty$, the CTS behind Coq. However, because these specifications have some specific 4structure, the problem is also decidable: In the case of $\mathscr{C}^C_\infty$, the problem is equivalent to a linear integer arithmetic problem[4]. Moreover, enumerating the functions is not scalable. In our implementation described in Chapter 10, we describe our implementation which uses an SMT solver to overcome this issue.

So far, we have proposed a correct algorithm to check whether a judgment derivable in a specification $\mathscr{C}$ can be derived in a specification $\mathscr{C}'$ via the notion of CTS embedding. This method is better than finding a specification morphism as already argued in Example 2.2 and 2.3. However, our method depends on the derivation tree built for this judgment. This might raise an issue if the derivation tree is not general enough as shown below in Example 2.10. Because the definition of a free CTS for a judgment $\Gamma \vdash_\mathscr{C} t : A$ depends on a derivation tree $\mathcal{T}$, to guarantee the completeness of our method, we need introduce in the next section *canonical* trees. The existence of such canonical tree for a judgment is not obvious and at this time is still an open problem. The section below discusses about the existence of such canonical tree and give some ideas on how it could be built.

[4]One could also note that empirically, most of the proofs written in Coq at this time do not use many universes, probably no more than 5 or most of the proofs ever written in Coq.

2.4 Completeness

The method we proposed previously relies on one particular derivation tree. This is not satisfactory for completeness, because we aim to decide whether a judgment is $(\mathscr{C},\mathscr{C}')$-embedded, not one particular derivation tree of this judgment. The example below shows that for the same judgment, our method returns two different results depending on the derivation chosen in the first place.

Example 2.10 *The judgment* $A : \star \vdash_{\mathbf{2}} \lambda x : A.\, x : A$ *admits the following derivation tree* Π_{bad}*:*

$$\dfrac{\dfrac{\Pi}{A : \star \vdash_{\mathbf{2}} \lambda x : A.\, x : (\lambda x : \star.\, x)\ A} \qquad \dfrac{\vdots}{A : \star \vdash_{\mathbf{2}} A : \star} \qquad \dfrac{}{(\lambda x : \star.\, x)\ A \equiv_\beta A}}{A : \star \vdash_{\mathbf{2}} \lambda x : A.\, x : A}$$

where Π *is*

$$\dfrac{\dfrac{\vdots}{A : \star \vdash_{\mathbf{2}} \lambda x : A.\, x : A} \qquad \dfrac{\vdots}{A : \star \vdash_{\mathbf{2}} (\lambda x : \star.\, x)\ A : \star} \qquad \dfrac{}{A \equiv_\beta (\lambda x : \star.\, x)\ A}}{A : \star \vdash_{\mathbf{2}} \lambda x : A.\, x : (\lambda x : \star.\, x)\ A}$$

One can prove that there is no specification morphism from $\mathscr{F}_{\Pi_{bad}}$ *to the* Simply Typed Lambda Calculus*. The reason is that the derivation tree above uses polymorphism and this will be reflected into the free CTS specification. At least, we can say that the free CTS contain the rule* $(?_i, ?_j, ?_k) \in \mathcal{R}_{\mathcal{F}_{\Pi_{bad}}}$ *and the axiom* $(?_j, ?_i) \in \mathcal{A}_{\mathcal{F}_{\Pi_{bad}}}$ *because of the expansion introduced by* Π*. However, a specification morphism from* $\mathscr{F}_{\Pi_{bad}}$ *to the specification* $\to$ *needs to map every product to* $(\star,\star,\star)$ *which implies to map the axiom* $?_j, ?_i$ *to* $(\star,\star)$ *which does not exist. However, this same judgment also admits the following derivation tree* Π_{good}*:*

$$\dfrac{\dfrac{\dfrac{\vdots}{A : \star, x : A \vdash_{\mathbf{2}} \mathbf{wf}}}{A : \star, x : A \vdash_{\mathbf{2}} x : A} \qquad \dfrac{\vdots}{A : \star \vdash_{\mathbf{2}} A \to A : \star}}{A : \star \vdash_{\mathbf{2}} \lambda x : A.\, x : A}$$

and now, we can see that the free CTS Π_{good} *is embeddable in the* Simply Typed Lambda Calculus*.*

Example 2.10 highlights that the choice of the derivation tree matters. However, the derivation tree we presented is pathological because it uses a β-expansion in an unnecessary way. So the completeness of the method relies on finding the appropriate derivation tree that we call here *canonical* derivation tree. Below, we explain a way to show the existence of such *canonical* tree and how to derive it. Our approach is to use an order on derivation trees defined as follows:

Definition 2.4.1 (Preorder for derivation trees)
Let $\mathcal{T}$ *and* $\mathcal{T}$*' two derivations of the same judgment, we define* $\mathcal{T} \preceq \mathcal{T}'$ *as*

$$\mathcal{T} \preceq \mathcal{T}' := \mathscr{F}_{\mathcal{T}} \trianglelefteq \mathscr{F}_{\mathcal{T}'}$$

Theorem 2.4.1 $\preceq$ *is a preorder.*

Proof *Direct consequence of Theorem 2.2.1.*

Example 2.11 *Using the derivation trees in Example 2.10, we can deduce that Π_{good} is smaller than Π_{bad} because it is easy to find a specification morphism from $\mathscr{F}_{\Pi_{good}}$ to $\mathscr{F}_{\Pi_{bad}}$.*

In general, we cannot expect $\preceq$ to be a total order. The reason for that is one can derive two derivation trees Π_l and Π_r for a same judgment such that Π_l uses dependent types and Π_r uses polymorphism for example (this could be done for Example 2.10). Hence, Π_l could be derived in the specification **P** and Π_r in the specification **2** for example. But obviously, there is no specification morphism between the two free CTS specifications generated by these derivation trees. However, we can expect that if there is a derivation tree which uses polymorphism but not dependent types and another which uses dependent types but not polymorphism, then there is one which does not use polymorphism nor dependent types. This is our first conjecture:

Conjecture 4 (Existence of a minimum for derivation tree preorder) *Given two derivation trees $\mathcal{T}$ and $\mathcal{T}'$ of the same judgment, there exist $\mathcal{T}''$ such that $\mathcal{T}'' \preceq \mathcal{T}$ and $\mathcal{T}'' \preceq \mathcal{T}'$.*

Definition 2.4.2 (Strict preorder for derivation tree)
The strict version $\mathcal{T} \prec \mathcal{T}'$ is defined as

$$\mathcal{T} \prec \mathcal{T}' := \mathscr{C}_{\mathcal{T}} \lhd \mathscr{C}_{\mathcal{T}'}$$

Example 2.12 *Going back to Example 2.10, we can check that $\Pi_{good} \prec \Pi_{bad}$ because the free CTS of Π_{good} cannot express polymorphism.*

Once we have defined a strict order, the natural question that follows is whether this strict order is well-founded. We cannot use the definition of $\lhd$ because the order on specifications is not well-founded in general. An example of that is Matita's specification (but it could be any specification with an infinite number of sorts).

Theorem 2.4.2 *$\lhd$ is not well-founded.*

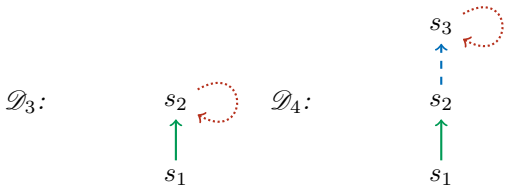
Proof *In the specification $\mathscr{C}^M_\infty$, one can create new specifications $\mathscr{C}^M_{\infty_i}$ for $i \in \mathbb{N}$ where the sorts $\{(n, P) \mid n \leq i\}$ are removed as well as all its dependencies in $\mathcal{A}$,$\mathcal{R}$ and $\mathcal{C}$. We clearly have $\mathscr{C}^M_{\infty_{i+1}} \lhd \mathscr{C}^M_{\infty_i}$, hence $\lhd$ is not well-founded.*

However, the argument that $\lhd$ is not well-founded cannot be applied for $\prec$ because all the specifications related by $\prec$ are finite.

Conjecture 5 (Derivation tree preordering is well-founded) *$\prec$ is well-founded.*

Neither can the proof rely on the number of sorts since we can have $\mathcal{T} \prec \mathcal{T}'$ but the free CTS of $\mathcal{T}$ have more sorts than the free CTS of $\mathcal{T}'$. Our argument in favor of this conjecture is related to the size of the derivation tree. Among all the derivation trees for a judgment, there is a smallest one. What makes the smallest derivation tree interesting is that adding $\mathscr{C}_{sort}, \mathscr{C}_\Pi$ never gives you more freedom to define a specification morphisms, it always generates more constraints. However, taking the minimal tree according to its size is not the best candidate for a canonical tree because of the rules $\mathscr{C}_{\preceq}$ and $\mathscr{C}^s_{\preceq}$. This is because using these rules in the derivation tree gives you more freedom to define a specification morphism as shown in the example below.

Example 2.13 *Let us define the CTS specifications $\mathscr{D}_3$ and $\mathscr{D}_4$ as follows:*



We can derive the judgment $\vdash_{\mathscr{D}_3} s_1 \to s_1 : s_2$ *with this derivation tree*[5]*:*

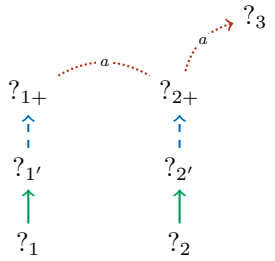
$$\dfrac{\dfrac{}{\vdash_{\mathscr{D}_3} s_1 : s_2} \quad \dfrac{}{\vdash_{\mathscr{D}_3} s_1 : s_2} \quad (s_2, s_2, s_2) \in \mathcal{R}_{\mathscr{D}_3}}{\vdash_{\mathscr{D}_3} s_1 \to s_1 : s_2}$$

which is clearly the minimal tree for this judgment. However, this tree is not $(\mathscr{D}_3, \mathscr{D}_4)$*-embeddable because the free specification generated for this tree has an empty cumulativity relation. However, this other derivation tree is* $(\mathscr{D}_3, \mathscr{D}_4)$*-embeddable.*

$$\dfrac{\dfrac{\dfrac{}{\vdash_{\mathscr{D}_3} s_1 : s_2} \quad s_2 \preceq_{\mathscr{C}} s_2}{\vdash_{\mathscr{D}_3} s_1 : s_2} \quad \dfrac{\dfrac{}{\blacksquare : s_1 \vdash_{\mathscr{D}_3} s_1 : s_2} \quad s_2 \preceq_{\mathscr{C}} s_2}{\blacksquare : s_1 \vdash_{\mathscr{D}_3} s_1 : s_2} \quad (s_2, s_2, s_2) \in \mathcal{R}_{\mathscr{D}_3}}{\vdash_{\mathscr{D}_3} s_1 \to s_1 : s_2}$$

The free CTS associated with this derivation tree is $\vdash_{\mathcal{F}_\mathcal{T}} ?_1 \to ?_2 : ?_3$ *where* $\mathcal{F}_\mathcal{T}$ *is defined as:*

$$(\mathcal{F}_\mathcal{T}) = \begin{cases} \mathcal{S} = & \{?_1, ?_{1'}, ?_{1+}, ?_2, ?_{2'}, ?_{2+}, ?_3\} \\ \mathcal{A} = & \{(?_1, ?_{1'}), (?_2, ?_{2'})\} \\ \mathcal{R} = & \{(?_{1+}, ?_{2+}, ?_3)\} \\ \mathcal{C} = & \{(?_{1'}, ?_{1+}), (?_{2'}, ?_{2+})\} \end{cases}$$



and there is a specification morphism from $\mathcal{F}_\mathcal{T}$ *to* $\mathscr{D}_4$ *as witnessed by the following function:*

$$\begin{array}{lll} ?_1 \to s_1 & ?_{1'} \to s_2 & ?_{1+} \to s_3 \\ ?_2 \to s_1 & ?_{2'} \to s_2 & ?_{2+} \to s_3 \\ & ?_3 \to s_3 & \end{array}$$

Hence, we have shown that the minimal tree, is not a canonical tree.

The example above shows that finding such canonical tree is not that easy because this tree may have unnecessary applications of the rules $\mathscr{C}_\preceq$ or $\mathscr{C}^s_\preceq$.[6] Assuming the two conjectures above, one can prove the existence of such canonical tree.

[5] We simplified the rule $\mathscr{C}_\Pi$ for non-dependent types.

[6] Actually, it is not clear whether these subtyping rules should be part of the derivation tree or it should be part of the definition of free CTS. In that case, the free CTS generated for the two derivation trees of Example 2.13 should be the same.

Theorem 2.4.3 *Given a judgment $\Gamma \vdash_{\mathscr{C}} t : A$, there exists a minimal tree $\mathcal{T}$, such that for all derivation tree $\mathcal{T}'$ of the same judgment $\mathcal{T} \preceq \mathcal{T}'$.*

Proof *A direct consequence of the conjectures 4 and 5.*

Following Example 2.13, the canonical tree should have subtyping rules whenever it is possible. Given a derivation tree, it should be possible to add one subtyping rule (either $\mathscr{C}_{\preceq}$ or $\mathscr{C}^{s}_{\preceq}$) between two rules which are not already a subtyping rule and to do that every time it is possible.

Conjecture 6 (Existence of minimal derivation tree) *A minimal derivation tree (according to its size) where unnecessary subtyping rules are added whenever possible is a canonical tree, meaning the smallest one for the order $\preceq$.*

In practice, all the type checkers are deterministic and given a judgment, compute one derivation tree which, from what we know seems minimal. While these algorithms do not introduce trivial expansions as in Example 2.10, they also do not introduce these trivial cumulativity rules as shown in Example 2.13 which are useful to find a specification morphism afterwards. In our work (see Chapter 10), we overcome this issue using the conjecture above: Using *explicit casts*, we add an *identity cast* wherever it could be useful, meaning applications for which the result type is either a sort or a product ending with a sort.

2.5 Future work

Another equivalence for CTS specifications: Our notion of CTS equivalence in Definition 2.1.3 and weak CTS equivalence in Definition 2.1.10 could be weakened so that we use a computational equality $\equiv_{\beta\star}$ instead of a syntactic fequality $=_{\star}$. The main advantage of this new definition is that reducing a term by β-reduction is more flexible for interoperability. For example the following judgment $A : \star \vdash_{\mathbf{P}} (\lambda x : \square.\, A)\, \mathbb{N} \to \star :$ is derivable in $\mathbf{P}$ but cannot be embedded in the $\to$ specification. However, we can apply a β-reduction and get the judgment $A : \star \vdash_{\mathbf{P}} A : \star$ which is embeddable in $\to$. This example suggests that taking the normal form of a term (if it exists) is always better for interoperability. However, this does not scale in practice. Moreover, during our experimentation, we observed that such generality was not useful. Finally, having conversion breaks the method to decide whether a judgment is $(\mathscr{C}, \mathscr{C}')$-embedded introduced in Section 2.3. It is not clear whether our method could be improved in an efficient way to take into account β-conversion.

Decidability of type checking for CTS: It would be interesting to see if this result could be used with a former result from Barthe in [Bar99b] where he shows that the type checking for any decidable, functional, injective and terminating PTS is decidable. Extending his result requires to include cumulativity in his algorithm and it is not clear how this could be achieved.

Chapter 3

Well-structured derivation trees

Our encoding of CTS into the $\lambda\Pi$-CALCULUS MODULO THEORY presented in Chapter 6 is an extension of Ali Assaf's results [Ass15b]. Investigating his correction proof leads us to realize that there was a subtle mistake which makes the proof erroneous. The only way we found to fix his proof is related to a famous conjecture formulated on PTS which can be extended on CTS: The equivalence between a presentation of CTS where the conversion is typed (*semantics* CTS) and the usual presentation where conversion is untyped (*syntactic* CTS). This problem was solved for PTS by Siles [Sil10] but remains a conjecture for CTS. Informally, this conjecture expresses any conversion (using the rules $\mathscr{C}_{\preceq}$ and $\mathscr{C}^{s}_{\preceq}$) which appears in a derivation tree can be replaced by a derivation tree which contains all the intermediate steps used to derive that two types A and B are convertible (as in Higher-Order Logic for example). This new derivation ensures that any intermediate step is also well-typed. Such derivation tree can then be analyzed through an induction proof which is used to derive models [Gog95b] [AC07]. The equivalence between the two systems, semantics CTS and syntactic CTS is not clear at all because there is a well-foundedness issue. As it will be explained in details in Section 3.3, a naive idea would be to use subject reduction to derive this new derivation tree for explicit conversion. The problem is that the proof of subject reduction creates new implicit conversions. Hence it is not clear that this process of replacing each implicit conversion by an explicit one with subject reduction is well-founded. The other way to prove this equivalence is to prove subject reduction first, directly on semantics CTS. But this time, the issue comes from the fact that the Product injectivity (1.4.2) property cannot be proven easily (see Lemma 3.3.4).

In this chapter, we introduce a predicate over derivation trees called *well-structured*. The goal of this predicate is to assert assumptions over the derivation tree so that the well-foundedness issue disappears using a notion of *levels* which provides a decreasing argument. In particular, our approach is purely syntactic and does not rely on the CTS specification. Another advantage of this method is to provide modular proofs between two equivalent type systems.

Then, we explore two alternative typing systems which are related to CTS. The first one removes the symmetry of $\equiv_\beta$ by only allowing reductions. The second typing system is the one we mentioned earlier which replaced the untyped conversion $\equiv_\beta$ with a typed conversion, in particular every β steps are typed. The first system is related to a famous conjecture called *Expansion Postponement* and the second was a conjecture for PTS and has been solved by Vincent Siles [Sil10] but remains a conjecture for CTS. In this chapter we prove that all these typing system are equivalent for well-structured derivation trees.

Finally, in the last section of this chapter, we open a discussion about well-structured derivation trees. In particular, we conjecture that every derivation tree is well-structured. In particular, we propose several typing systems which aim at providing criteria showing that some classes of

derivation trees are well-structured.

3.1 Well-structured derivation trees

One of the main results in the meta-theory of CTS is Subject Reduction (1.7.11). It is proven using an another lemma: the Substitution Lemma (1.7.8). However, in the case of our embedding in Chapter 6 (and the Expansion Postponement conjecture, see Section 3.2), this dependency exists in the other way around which introduces a cycle. This non-desired dependencies appear in the proof of the substitution lemma for non *structural* rules, namely $\mathscr{C}_\lambda$ and $\mathscr{C}_\preceq$: In those two rules, one of the premises ensures that the type is well-sorted. The reason for this new dependency is that while proving the substitution lemma for t in $\Gamma \vdash_{\mathscr{C}} t : A$, we also need to have the subject reduction property for the type A when the rule is $\mathscr{C}_\lambda$ for example. To remove this circularity, we introduce the notion of *well-structured derivation trees* below.

The purpose of *well-structured derivation trees* is to find a decreasing argument to the circularity issue we have mentioned previously. The central idea is to have a *level* for derivation trees which strictly decreases when we go from the derivation of a term to the derivation of its type and is stable by β-reduction.

Definition 3.1.1 (Subtree, has-type, and β relations for derivation trees)
We define the three following relations on derivation trees:

- $\pi \lhd \pi'$ *if* π *is a subtree of* π'
- $\pi \prec \pi'$ *if* $\dfrac{\pi}{\Gamma \vdash_{\mathscr{C}} A : s}$ *and* $\dfrac{\pi'}{\Gamma \vdash_{\mathscr{C}} t : A}$
- $\pi \hookrightarrow_\beta \pi'$ *if* $\dfrac{\pi}{\Gamma \vdash_{\mathscr{C}} t : A}$ *and* $\dfrac{\pi'}{\Gamma \vdash_{\mathscr{C}} t' : A}$ *with* $t \hookrightarrow_\beta t'$

The general idea behind well-structured derivation trees is to generate an induction principle which is compatible with subject reduction. The usual way to compute the β reduct of a derivation tree is to use the subject reduction lemma as a computable function. However, subject reduction may increase the size of the derivation tree and therefore is not compatible with the usual induction principle on trees.

Our idea is to find a measure which strictly decreases when we go from a term to its type (via the relation $\prec$) and which does not increase via subject reduction.

Given a CTS derivation tree, the relation $\prec$ can be turned into a computing function using the Lemma 1.7.9 and so does for the relation $\hookrightarrow_\beta$ using the Subject Reduction Lemma 1.7.11. However, it is not clear whether the proofs (seen as computable functions) we have are compatible with the intuitions described above and this is why the definition of well-structured derivation trees is parameterized by these two functions.

Definition 3.1.2 (Well-structured derivation tree)
A derivation tree π_1 *is said* well-structured *if there exists two functions* HT, SR *and a family* $(\mathcal{L}_n)_{n\in\mathbb{N}}$ *such that:*

$$\exists n,\ \pi_1 \in \mathcal{L}_n \qquad (WS_n)$$
$$\forall i,\ \mathcal{L}_i \subseteq \mathcal{L}_{i+1} \qquad (WS_\subseteq)$$
$$\forall i, \pi, \pi',\ \pi' \lhd \pi \wedge \pi \in \mathcal{L}_i \Rightarrow \pi' \in \mathcal{L}_i \qquad (WS_\lhd)$$
$$\forall i, \pi, \pi',\ \pi' = HT(\pi) \wedge \pi' \prec \pi \wedge \pi \in \mathcal{L}_{i+1} \Rightarrow \pi' \in \mathcal{L}_i \qquad (WS_\prec)$$
$$\forall i, \pi, \pi',\ \pi' = SR(\pi) \wedge \pi \hookrightarrow_\beta \pi' \wedge \pi \in \mathcal{L}_i \Rightarrow \pi' \in \mathcal{L}_i \qquad (WS_{\hookrightarrow_\beta})$$

Except stated otherwise, the functions HT and SR will be the usual ones. If a tree is well-structured, it is denoted $WS(\pi')$. *If* $\pi \in \mathcal{L}_i$, *then* i *is called a level and we said that* π *is derivable at level* i.

The first condition ensure that π_1 is at some level n. The second condition ensures that levels are cumulative. The third condition ensures that levels are closed by subtrees. The fourht condition ensure that if I can derive $\Gamma \vdash_{\mathscr{C}} t : A$ at som level $n+1$ then I can derive $\Gamma \vdash_{\mathscr{C}} A$ **ws** at level n. Finally the last condition ensure that if I can derive $\Gamma \vdash_{\mathscr{C}} t : A$ at level n, I can also derive $\Gamma \vdash_{\mathscr{C}} t' : A$ at level n where $t \hookrightarrow_\beta t'$.

A direct consequence of this definition is the following lemma.

Lemma 3.1.1 *If* $WS(\pi)$ *then:* $\forall \pi'$ *such that* $\pi' \lhd \pi$ *then* $WS(\pi')$.

As we will see in Section 3.4 the difficult part to show that a derivation tree is well-structured comes mostly from the last condition $WS_{\hookrightarrow_\beta}$.

Example 3.1 *Given a derivation tree of some judgment. If all the terms that appear in this derivation never contain a* β *redex, then the derivation tree is well-structured. By induction on the derivation tree, one may assign a level to all the subtrees, starting from* 0 *for the leaves. This level allows us to reconstruct the family* $(\mathcal{L}_n)_{n\in\mathbb{N}}$ *easily. The condition* $WS_{\hookrightarrow_\beta}$ *is true by assumption.*

Example 3.2 *Any derivation tree which can be derived in the* Simply Typed Lambda Calculus $PTS(\rightarrow)$ *is well-structured. One may stratify terms of the* Simply Typed Lambda Calculus *as: sorts, types and terms. Any derivation tree of a sort is derivable at level* 0, *of a type at level* 1 *and a term at level* 2. *Because in the* Simply Typed Lambda Calculus *a type cannot contain any* β *redex, the level of a sort and a type of* Simply Typed Lambda Calculus *is stable by* β *reduction automatically. We can therefore conclude that any term is stable by* β *reduction at level* 2. *This proof is made formal in Theorem 3.4.3.*

These examples quite informal shows that there exist derivation trees which are well-structured. Can we found derivation trees which are **not** well-structured? We conjecture that such derivation tree does not exists and is an open question discussed in Section 3.4.

While writing proofs, we prefer to manipulate anonymously derivation trees by writing *a derivable judgment* than giving a name to the derivation tree. To reflect this with well-structured derivation trees, we define below well-structured judgments.

Definition 3.1.3 (Well-structured judgments)
A judgment $\Gamma \vdash_{\mathscr{C}} t : A$ *or* $\Gamma \vdash_{\mathscr{C}}$ **wf** *is* well-structured *if it is derivable by a well-structured derivation tree.*

Notation 21 *A well-structured judgment will be denoted* $WS(\Gamma \vdash_{\mathscr{C}} t : A)$ *or* $WS(\Gamma \vdash_{\mathscr{C}}$ **wf**$)$. *A well-structured judgment at level* n *will be denoted* $WS_n(\Gamma \vdash_{\mathscr{C}} t : A)$ *or* $WS_n(\Gamma \vdash_{\mathscr{C}}$ **wf**$)$. *This notation is also extended naturally for* $\Gamma \vdash_{\mathscr{C}} A$ **ws** *and will be denoted* $WS_n(\Gamma \vdash_{\mathscr{C}} A$ **ws**$)$.

Lemma 3.1.2 *Any inversion lemma as stated in Section 1.7 preserves the well-structured property.*

Proof *This is a direct consequence of the well-structured judgment definition and Lemma 3.1.1.*

One interesting lemma that we will use in next section is the following one.

Lemma 3.1.3 *If* $WS_{n+1}(\Gamma \vdash_{\mathscr{C}} t : A)$, $WS_n(\Gamma \vdash_{\mathscr{C}} B : s)$ *and* $A \equiv_\beta B$ *then* $WS_{n+1}(\Gamma \vdash_{\mathscr{C}} t : B)$.

$$\frac{A \hookrightarrow_\beta^* B}{A \preceq_{\mathscr{C}}^r B} \preceq_{\equiv_\beta}^r \qquad \frac{(s, s') \in \mathcal{C}_{\mathscr{C}}^*}{s \preceq_{\mathscr{C}}^r s'} \preceq_{\mathcal{C}_{\mathscr{C}}}^r \qquad \frac{A \hookrightarrow_\beta^* A' \qquad B \preceq_{\mathscr{C}}^r B'}{(x:A) \to B \preceq_{\mathscr{C}}^r (x:A') \to B'} \preceq_{\Pi}^r$$

Figure 3.1: CTS subtyping relation with reduction only

3.2 Expansion postponement

Expansion postponement is a conjecture which breaks the symmetry of the conversion in CTS. It expresses that any derivable judgment can be derived in a type system where the conversion steps are restricted to reductions steps except at the end of the derivation.

Definition 3.2.1 (Typing system without expansions)
We denote $\Gamma \vdash_{\mathscr{C}^r}^t t : A$ the type system where the conversion rules $\mathscr{C}_\preceq$ and $\mathscr{C}_\preceq^s$ are restricted to reductions only. It is presented in Fig. 3.1 and Fig. 3.2

The Expansion Postponement conjecture is generally stated as follows:

Conjecture 7 (Expansion postponement) *If $\Gamma \vdash_{\mathscr{C}} t : A$ then $\Gamma \vdash_{\mathscr{C}^r}^t t : A'$ where $A \hookrightarrow_\beta^* A'$.*

The problem to prove this conjecture comes from the rule $\mathscr{C}_\lambda$ because the type B in the left premise, occurs as a subject of the right premise $(x:A) \to B$. We put this problem in evidence with the example below.

Example 3.3 *In the specification* **P** *(LF), we define the typing context Γ as:*

- $\mathbb{N} : \star$
- $Vect : \mathbb{N} \to \star$
- $f : (x:\mathbb{N}) \to Vect\ x$

Assuming we have a derivation of $\Gamma \vdash_{\mathscr{C}} t : \mathbb{N} \to \mathbb{N}$ and that $t \hookrightarrow_\beta t'$. Then one can derive that $\Gamma \vdash_{\mathscr{C}} \lambda x:\mathbb{N}.\, f\ (t\ x) : (x:\mathbb{N}) \to Vect\,(t\ x)$. Using subject reduction, one can also prove that we have $\Gamma \vdash_{\mathscr{C}} \lambda x:\mathbb{N}.\, f\ (t'\ x) : (x:\mathbb{N}) \to Vect\,(t\ x)$. However, this derivation tree will contain an expansion (if we follow the usual proof of subject reduction): We have $\Gamma, x : \mathbb{N} \vdash_{\mathscr{C}} f\ (t'\ x) : Vect\,(t'\ x)$ and we want to derive that $\Gamma, x : \mathbb{N} \vdash_{\mathscr{C}} f\ (t'\ x) : Vect\,(t\ x)$. Hence we introduce an expansion using the rule $\mathscr{C}_\preceq$ because $t'\ x \hookleftarrow_\beta^* t\ x$. Then we can conclude with the rule $\mathscr{C}_\lambda$. The expansion postponement conjecture says that the expansions we have introduced with the rule $\mathscr{C}_\preceq$ can be switched with the rule $\mathscr{C}_\lambda$. To do so, we have to give a proof that $(x:\mathbb{N}) \to Vect\,(t'\ x)$ is well-sorted using only reduction rules. Such proof generally relies on subject reduction. Subject reduction for $\Gamma \vdash_{\mathscr{C}^r}^t t : A$ typing system is hard to prove without strong hypothesis on the specification.

Well-structured derivation trees give a way to solve this problem by doing first an induction on the level: If the derivation tree of $\Gamma \vdash_{\mathscr{C}} \lambda x:\mathbb{N}.\, f\ (t\ x) : (x:\mathbb{N}) \to Vect\,(t\ x)$ is well-structured, then there exist n such that $\Gamma \vdash_{\mathscr{C}} \lambda x:\mathbb{N}.\, f\ (t\ x) : (x:\mathbb{N}) \to Vect\,(t\ x)$ is derivable at level $n+1$ and $\Gamma \vdash_{\mathscr{C}} t : \mathbb{N} \to \mathbb{N}$ is derivable at level n by $WS_\preceq$. By the well-structured hypothesis $WS_{\hookrightarrow_\beta}$, we can derive that $\Gamma \vdash_{\mathscr{C}} t' : \mathbb{N} \to \mathbb{N}$ is derivable at level n. Hence, we see that proving subject reduction at level n solves the abstraction cases at level $n+1$.

$$\frac{}{\varnothing \vdash_{\mathscr{C}} \mathbf{wf}}\ \mathscr{C}^{\mathbf{wf}^r}_{\varnothing}$$

$$\frac{\Gamma \vdash_{\mathscr{C}} A : s \qquad x \notin \Gamma}{\Gamma, x : A \vdash_{\mathscr{C}} \mathbf{wf}}\ \mathscr{C}^{\mathbf{wf}^r}_{var}$$

$$\frac{\Gamma \vdash_{\mathscr{C}} \mathbf{wf} \qquad (x : A) \in \Gamma}{\Gamma \vdash_{\mathscr{C}} x : A}\ \mathscr{C}^{r}_{var}$$

$$\frac{\Gamma \vdash_{\mathscr{C}} \mathbf{wf} \qquad (s_1, s_2) \in \mathcal{A}}{\Gamma \vdash_{\mathscr{C}} s_1 : s_2}\ \mathscr{C}^{r}_{sort}$$

$$\frac{\Gamma \vdash_{\mathscr{C}} A : s_1 \qquad \Gamma, x : A \vdash_{\mathscr{C}} B : s_2 \qquad (s_1, s_2, s_3) \in \mathcal{R}}{\Gamma \vdash_{\mathscr{C}} (x : A) \to B : s_3}\ \mathscr{C}^{r}_{\Pi}$$

$$\frac{\Gamma, x : A \vdash_{\mathscr{C}} M : B \qquad \Gamma \vdash_{\mathscr{C}} (x : A) \to B : s}{\Gamma \vdash_{\mathscr{C}} \lambda x : A.\, M : (x : A) \to B}\ \mathscr{C}^{r}_{\lambda}$$

$$\frac{\Gamma \vdash_{\mathscr{C}} M : (x : A) \to B \qquad \Gamma \vdash_{\mathscr{C}} N : A}{\Gamma \vdash_{\mathscr{C}} M\ N : B\{x \leftarrow N\}}\ \mathscr{C}^{r}_{app}$$

$$\frac{\Gamma \vdash_{\mathscr{C}} M : A \qquad \Gamma \vdash_{\mathscr{C}} B : s \qquad A \preceq_{\mathscr{C}^r} B}{\Gamma \vdash_{\mathscr{C}} M : B}\ \mathscr{C}^{r}_{\preceq}$$

$$\frac{\Gamma \vdash_{\mathscr{C}} M : A \qquad A \preceq_{\mathscr{C}^r} s}{\Gamma \vdash_{\mathscr{C}} M : s}\ \mathscr{C}^{s^r}_{\preceq}$$

Figure 3.2: Typing rules for CTS using only reductions

The example above shows how the well-structured hypothesis can be used as an induction scheme over typing derivations. This is detailed in the definition below.

Definition 3.2.2 (EP)
We define the expansion postponement property at level n (EP_n) as:

- *If $WS_n(\Gamma \vdash_{\mathscr{C}} t : A)$ then we can derive $\Gamma \vdash^t_{\mathscr{C}^r} t : A'$ where $A \hookrightarrow^*_\beta A'$*
- *If $WS_n(\Gamma \vdash_{\mathscr{C}} \mathbf{wf})$ then we can derive $\Gamma \vdash_{\mathscr{C}^r} \mathbf{wf}$*

We define EP as for all n, EP_n.

Lemma 3.2.1 *If $\Gamma \vdash^t_{\mathscr{C}^r} t : A$ then $\Gamma \vdash_{\mathscr{C}} t : A$. If $\Gamma \vdash_{\mathscr{C}^r} \mathbf{wf}$ then $\Gamma \vdash_{\mathscr{C}} \mathbf{wf}$.*

Proof *By induction on the derivation. Trivial since $\mathscr{C}^r_{\preceq}$ and $\mathscr{C}^{s^r}_{\preceq}$ are restrictions of $\mathscr{C}_{\preceq}$ and $\mathscr{C}^s_{\preceq}$.*

Lemma 3.2.2 *If $A \preceq_{\mathscr{C}} B$ then $A \preceq_{\mathscr{C}^r} B'$ where $B \hookrightarrow^*_\beta B'$.*

Proof *By induction on* $A \underline{\preceq}_{\mathscr{C}} B$.

The lemma below is the induction step we have mentioned in Example 3.3.

Lemma 3.2.3 $EP_n \Rightarrow EP_{n+1}$.

Proof *By induction on the typing derivation. All the inversions lemmas use implicitly Lemma (3.1.2). Moreover, the proof below does not handle the easy cases where* $\Gamma \vdash_{\mathscr{C}} t : A$ *and* $A \in \mathcal{S}^{\top}_{\mathscr{C}}$ *meaning that* A *is a top-sort for readability. One can check that for the cases* $\mathscr{C}_{app}$, $\mathscr{C}_{\preceq}$ *and* $\mathscr{C}^{s}_{\preceq}$ *where this assumption is made, the cases can be closed easily.*

◊ $\mathscr{C}^{\mathbf{wf}}_{\varnothing}$:

Trivial.

◊ $\mathscr{C}^{\mathbf{wf}}_{var}$: $\Gamma = \Gamma', x : A$

	(1)	$WS_{n+1}(\Gamma', x : A \vdash_{\mathscr{C}} \mathbf{wf})$	*Main Hypothesis*	
	(2)	$WS_{n+1}(\Gamma' \vdash_{\mathscr{C}} A : s)$	*Inversion on* $\mathscr{C}^{\mathbf{wf}}_{var}$	1
	(3)	$\Gamma' \vdash^{t}_{\mathscr{C}^r} A : s$	*Induction Hypothesis*	2
✯	(4)	$\Gamma', x : A \vdash_{\mathscr{C}^r} \mathbf{wf}$	$\mathscr{C}^{\mathbf{wf}^r}_{var}$	3

◊ $\mathscr{C}_{var}$: $t = x$

	(1)	$WS_{n+1}(\Gamma \vdash_{\mathscr{C}} x : A)$	*Main Hypothesis*	
	(2)	$WS_{n+1}(\Gamma \vdash_{\mathscr{C}} \mathbf{wf})$	*Inversion on* $\mathscr{C}_{var}$	1
	(3)	$\Gamma \vdash_{\mathscr{C}^r} \mathbf{wf}$	*Induction Hypothesis*	2
✯	(4)	$\Gamma \vdash^{t}_{\mathscr{C}^r} x : A$	$\mathscr{C}^{r}_{var}$	3

◊ $\mathscr{C}_{sort}$: $t = s, A = s'$

	(1)	$WS_{n+1}(\Gamma \vdash_{\mathscr{C}} s : s')$	*Main Hypothesis*	
	(2)	$WS_{n+1}(\Gamma \vdash_{\mathscr{C}} \mathbf{wf})$	*Inversion on* $\mathscr{C}_{sort}$	1
	(3)	$\Gamma \vdash_{\mathscr{C}^r} \mathbf{wf}$	*Induction Hypothesis*	2
✯	(4)	$\Gamma \vdash^{t}_{\mathscr{C}^r} s : s'$	$\mathscr{C}^{r}_{sort}$	3

◊ $\mathscr{C}_{\Pi}$: $t = (x : C) \to D, A = s$

	(1)	$WS_{n+1}(\Gamma \vdash_{\mathscr{C}} (x : C) \to D : s)$	*Main Hypothesis*	
	(2)	$WS_{n+1}(\Gamma \vdash_{\mathscr{C}} C : s_1)$	*Inversion on* $\mathscr{C}_{\Pi}$	1
	(3)	$WS_{n+1}(\Gamma, x : C \vdash_{\mathscr{C}} D : s_2)$		
	(4)	$(s_1, s_2, s) \in \mathcal{R}$		
	(5)	$\Gamma \vdash^{t}_{\mathscr{C}^r} C : s_1$	*Induction Hypothesis*	2
	(6)	$\Gamma, x : C \vdash^{t}_{\mathscr{C}^r} D : s_2$	*Induction Hypothesis*	3
✯	(7)	$\Gamma \vdash^{t}_{\mathscr{C}^r} (x : C) \to D : s$	$\mathscr{C}^{r}_{\Pi}$	5,6,4

◊ $\mathscr{C}_{\lambda}$: $t = \lambda x : C.\, t_1, A = (x : C) \to D$

	(1)	$WS_{n+1}(\Gamma \vdash_{\mathscr{C}} \lambda x : C.\, t_1 : (x : C) \to D)$	*Main Hypothesis*	
	(2)	EP_n		
	(3)	$WS_{n+1}(\Gamma, x : C \vdash_{\mathscr{C}} t_1 : D)$	*Inversion on* $\mathscr{C}_{\lambda}$	1
	(4)	$WS_n(\Gamma \vdash_{\mathscr{C}} (x : C) \to D : s)$		
	(5)	$\Gamma, x : C \vdash^{t}_{\mathscr{C}^r} t_1 : D'$	*Induction Hypothesis*	3
	(6)	$D \hookrightarrow^{*}_{\beta} D'$		
	(7)	$(x : C) \to D \hookrightarrow^{*}_{\beta} (x : C) \to D'$	*Congruence of* β	6
	(8)	$\Gamma \vdash^{t}_{\mathscr{C}^r} (x : C) \to D' : s$	EP_n	2,4
✯	(9)	$\Gamma \vdash^{t}_{\mathscr{C}^r} \lambda x : C.\, t_1 : (x : C) \to D$	$\mathscr{C}^{r}_{\lambda}$	5,8

◊ $\mathscr{C}_{app}$: $t = t_1\ t_2, A = D\{x \leftarrow t_2\}$

	(1)	$WS_{n+1}(\Gamma \vdash_{\mathscr{C}} t_1\ t_2 : C\{x \leftarrow t_2\})$	*Main Hypothesis*	
	(2)	EP_n		
	(3)	$WS_{n+1}(\Gamma \vdash_{\mathscr{C}} t_1 : (x:C) \to D)$	*Inversion on* $\mathscr{C}_{app}$	1
	(4)	$WS_{n+1}(\Gamma \vdash_{\mathscr{C}} t_2 : C)$		
	(5)	$\Gamma \vdash^t_{\mathscr{C}^r} t_1 : (x:C') \to D'$	*Induction Hypothesis*	3
	(6)	$(x:C) \to D \hookrightarrow^*_\beta (x:C') \to D'$		
	(7)	$C \hookrightarrow^*_\beta C'$	*Product injectivity (1.4.2)*	6
	(8)	$D \hookrightarrow^*_\beta D'$		6
	(9)	$\Gamma \vdash^t_{\mathscr{C}^r} t_2 : C''$	*Induction Hypothesis*	4
	(10)	$C \hookrightarrow^*_\beta C''$		
	(11)	$C' \hookrightarrow^*_\beta C''' \hookleftarrow^*_\beta C''$	*Confluence of* β	7,10
	(12)	$WS_n(\Gamma \vdash_{\mathscr{C}} C : s)$	$WS_\prec$	4
	(13)	$WS_n(\Gamma \vdash_{\mathscr{C}} C''' : s)$	$WS_{\hookrightarrow_\beta}$	12,10,11
	(14)	$\Gamma \vdash^t_{\mathscr{C}^r} C''' : s$	EP_n	2,13
	(15)	$(x:C') \to D \hookrightarrow^*_\beta (x:C''') \to D$	*Congruence of* β	11
	(16)	$WS_n(\Gamma \vdash_{\mathscr{C}} (x:C) \to D : s')$	$WS_\prec$	3
	(17)	$WS_n(\Gamma \vdash_{\mathscr{C}} (x:C''') \to D' : s')$	$WS_{\hookrightarrow_\beta}$	16,10,11
	(18)	$\Gamma \vdash^t_{\mathscr{C}^r} (x:C''') \to D : s'$	EP_n	2,17
	(19)	$\Gamma \vdash^t_{\mathscr{C}^r} t_1 : (x:C''') \to D$	$\mathscr{C}_{\preceq}$	5,17,15
	(20)	$\Gamma \vdash^t_{\mathscr{C}^r} t_2 : C'''$	$\mathscr{C}_{\preceq}$	9,14,11
✯	(21)	$\Gamma \vdash^t_{\mathscr{C}^r} t_1\ t_2 : C'''\{x \leftarrow t_2\}$	$\mathscr{C}^r_{app}$	19,20

◊ $\mathscr{C}_{\preceq}$:

	(1)	$WS_{n+1}(\Gamma \vdash_{\mathscr{C}} t : A)$	*Main Hypothesis*	
	(2)	EP_n		
	(3)	$\Gamma \vdash_{\mathscr{C}} t : B$	*Inversion on* $\mathscr{C}_{\preceq}$	1
	(4)	$\Gamma \vdash_{\mathscr{C}} B : s$		
	(5)	$B \underline{\preceq}_{\mathscr{C}} A$		
	(6)	$\Gamma \vdash^t_{\mathscr{C}^r} t : B'$	*Induction Hypothesis*	3
	(7)	$B \hookrightarrow^*_\beta B'$		
	(8)	$WS_n(\Gamma \vdash_{\mathscr{C}} B' : s)$	$WS_\prec$	2,3,4
	(9)	$\Gamma \vdash^t_{\mathscr{C}^r} B' : s$	EP_n	2,8
	(10)	$B' \underline{\preceq}_{\mathscr{C}^r} A'$	*Lemma (3.2.2)*	7,5
✯	(11)	$\Gamma \vdash^t_{\mathscr{C}^r} t : A'$	$\mathscr{C}^r_{\preceq}$	6,9,10

◊ $\mathscr{C}^s_{\preceq}$: $A = s$

	(1)	$WS_{n+1}(\Gamma \vdash_{\mathscr{C}} t : A)$	*Main Hypothesis*	
	(2)	$WS_{n+1}(\Gamma \vdash_{\mathscr{C}} t : B)$	*Inversion on* $\mathscr{C}^s_{\preceq}$	1
	(3)	$B \underline{\preceq}_{\mathscr{C}} s$		
	(4)	$\Gamma \vdash^t_{\mathscr{C}^r} t : B'$	*Induction Hypothesis*	2
	(5)	$B \hookrightarrow^*_\beta B'$		
	(6)	$WS_n(\Gamma \vdash_{\mathscr{C}} B : s)$	$WS_{\hookrightarrow_\beta}$	2,5
	(7)	$B' \underline{\preceq}_{\mathscr{C}^r} s$	*Lemma (3.2.2)*	5,3
✯	(8)	$\Gamma \vdash^t_{\mathscr{C}^r} t : s$	$\mathscr{C}^r_{\preceq}$	4,7

Theorem 3.2.4 *For all* n, *we have* EP_n. *In particular we have* $WS_n(\Gamma \vdash_{\mathscr{C}} t : A)$ *then* $\Gamma \vdash^t_{\mathscr{C}^r} t : A$.

Proof *By induction on n.*

◊ n is a minimal element*:*

Only the rules $\mathscr{C}^{\mathbf{wf}}_{\varnothing}$, $\mathscr{C}^{\mathbf{wf}}_{var}$, $\mathscr{C}_{sort}$ and $\mathscr{C}^{s}_{\preceq}$ are possible. Otherwise it contradicts $WS_{\prec}$.

◊ Inductive case*:*

This is handled by Lemma 3.2.3.

In conclusion of this section, we have shown that any well-structured judgment satisfies also the expansion postponement property. Therefore, it also proves that if our conjecture 9 that any derivation-tree is well-structured is true, then we would solve the expansion postponement conjecture. At this time, we were not able to derive any result about the opposition direction. The problem is that the circularity we break using well-structured derivation trees can, a priori, still apply for derivation trees that do not use expansions. Breaking this circularity would require a deeper understanding of the meta-theory of this typing system, which we do not have at this time (except for specific CTS specification classes of course).

3.3 Semantic CTS

In this section, we will have a look at the explicit conversion system for CTS, i.e. *semantics* CTS. The idea is to define a new type system–semantic type system–such that the conversion $A \preceq_{\mathscr{C}} B$ becomes a judgment $\Gamma \vdash^{e}_{\mathscr{C}} A \preceq_{\mathscr{C}} B : s$ which details how A is a subtype of B and which gives a type to all the intermediate terms. Vincent Siles in [Sil10] proved the equivalence between the two versions for PTS but this question remains open for CTS. We prove in this section that for well-structured derivation trees the equivalence between the syntactic type system and the semantic one.

Definition 3.3.1 (Typing system with explicit subtyping)
We define the typing judgments $\Gamma \vdash^{e}_{\mathscr{C}} t : A$, $\Gamma \vdash^{e}_{\mathscr{C}} A \preceq_{\mathscr{C}} B : s$ and $\Gamma \vdash^{e}_{\mathscr{C}} A \equiv_{\beta} B : s$ in Fig. 3.3, Fig 3.4, and Fig 3.5. Since a top-sort might not have a type, we add in the syntax a special sort s_∞. Hence, for the judgment $\Gamma \vdash^{e}_{\mathscr{C}} A \preceq_{\mathscr{C}} B : s$, we have $s \in \mathcal{S}_{\mathscr{C}} \cup \{s_\infty\}$. By inversion, if $\Gamma \vdash^{e}_{\mathscr{C}} t : A$ then we can ensure that $A \neq s_\infty$ since the sort used to type $A \preceq_{\mathscr{C}} B$ is never used afterwards.

The judgment $\Gamma \vdash^{e}_{\mathscr{C}} A \equiv_{\beta} B : s$ is the same as defined by Vincent Siles in [Sil10]. We can notice an asymmetry in some rules such as $\mathscr{C}^{\equiv_\beta}_{\lambda}$ or $\mathscr{C}^{\equiv_\beta}_{\mathrm{app}}$. This is not an issue and it is discussed in [Sil10]. Adding s_∞ introduces another asymmetry in the rule $\preceq^{e}_{tr}$: We want to derive $\Gamma \vdash^{e}_{\mathscr{C}} s_1 \preceq_{\mathscr{C}} s_2 : s_\infty$, but if s_2 is the subtype of another sort s_3 which has a type s, we also want to derive $\Gamma \vdash^{e}_{\mathscr{C}} s_1 \preceq_{\mathscr{C}} s_3 : s$. Roughly, $\Gamma \vdash^{e}_{\mathscr{C}} A \preceq_{\mathscr{C}} B : s$ is just an extension of $\Gamma \vdash^{e}_{\mathscr{C}} t \equiv_{\beta} u : A$ with subtyping. As for expansion postponement, one direction is obvious:

Lemma 3.3.1 *The following implications hold:*

- *If $\Gamma \vdash^{e}_{\mathscr{C}} t : A$ then $\Gamma \vdash_{\mathscr{C}} t : A$*
- *If $\Gamma \vdash^{e}_{\mathscr{C}}$ $\mathbf{wf}$ then $\Gamma \vdash_{\mathscr{C}}$ $\mathbf{wf}$*
- *If $\Gamma \vdash^{e}_{\mathscr{C}} A \preceq_{\mathscr{C}} B : s$ then $\mathcal{A}_{\preceq_{\mathscr{C}}} B$*
- *If $\Gamma \vdash^{e}_{\mathscr{C}} A \equiv_{\beta} B : s$ then $A \equiv_{\beta} B$, $\Gamma \vdash_{\mathscr{C}} A : s$ and $\Gamma \vdash_{\mathscr{C}} B : s$*

Proof *By induction on the derivation.*

$$\frac{}{\varnothing \vdash^e_{\mathscr{C}} \mathbf{wf}}\ \mathscr{C}^{\mathbf{wf}}_{\varnothing}$$

$$\frac{\Gamma \vdash^e_{\mathscr{C}} A : s \qquad x \notin \Gamma}{\Gamma, x : A \vdash^e_{\mathscr{C}} \mathbf{wf}}\ \mathscr{C}^{\mathbf{wf}}_{var}$$

$$\frac{\Gamma \vdash^e_{\mathscr{C}} \mathbf{wf} \qquad (x : A) \in \Gamma}{\Gamma \vdash^e_{\mathscr{C}} x : A}\ \mathscr{C}_{var}$$

$$\frac{\Gamma \vdash^e_{\mathscr{C}} \mathbf{wf} \qquad (s_1, s_2) \in \mathcal{A}}{\Gamma \vdash^e_{\mathscr{C}} s_1 : s_2}\ \mathscr{C}_{sort}$$

$$\frac{\Gamma \vdash^e_{\mathscr{C}} A : s_1 \qquad \Gamma, x : A \vdash^e_{\mathscr{C}} B : s_2 \qquad (s_1, s_2, s_3) \in \mathcal{R}}{\Gamma \vdash^e_{\mathscr{C}} (x : A) \to B : s_3}\ \mathscr{C}_{\Pi}$$

$$\frac{\Gamma, x : A \vdash^e_{\mathscr{C}} M : B \qquad \Gamma \vdash^e_{\mathscr{C}} (x : A) \to B : s}{\Gamma \vdash^e_{\mathscr{C}} \lambda x : A.\, M : (x : A) \to B}\ \mathscr{C}_{\lambda}$$

$$\frac{\Gamma \vdash^e_{\mathscr{C}} M : (x : A) \to B \qquad \Gamma \vdash^e_{\mathscr{C}} N : A}{\Gamma \vdash^e_{\mathscr{C}} M\ N : B\{x \leftarrow N\}}\ \mathscr{C}_{app}$$

$$\frac{\Gamma \vdash^e_{\mathscr{C}} M : A \qquad \Gamma \vdash^e_{\mathscr{C}} B : s \qquad \Gamma \vdash^e_{\mathscr{C}} A \preceq_{\mathscr{C}} B : s}{\Gamma \vdash^e_{\mathscr{C}} M : B}\ \mathscr{C}_{\preceq}$$

$$\frac{\Gamma \vdash^e_{\mathscr{C}} M : A \qquad \Gamma \vdash^e_{\mathscr{C}} A \preceq_{\mathscr{C}} s : s_\infty}{\Gamma \vdash^e_{\mathscr{C}} M : s}\ \mathscr{C}^{s}_{\preceq}$$

Figure 3.3: Typing rules for annotated CTS

Theorem 3.3.2 (Vincent Siles [Sil10]) *If $\mathcal{C}_{\mathscr{C}} = \emptyset$ then*

$$\Gamma \vdash_{\mathscr{C}} t : A \Leftrightarrow \Gamma \vdash^e_{\mathscr{C}} t : A$$
$$\Gamma \vdash_{\mathscr{C}} \mathbf{wf} \Leftrightarrow \Gamma \vdash^e_{\mathscr{C}} \mathbf{wf}$$

The natural extension of this result to CTS gives the following conjecture.

Conjecture 8 (Equivalence between syntaxic and semantic CTS) *We conjecture the following equivalences:*

$$\Gamma \vdash_{\mathscr{C}} t : A \Leftrightarrow \Gamma \vdash^e_{\mathscr{C}} t : A$$
$$\Gamma \vdash_{\mathscr{C}} \mathbf{wf} \Leftrightarrow \Gamma \vdash^e_{\mathscr{C}} \mathbf{wf}$$

The theorem 3.3.2 is not easy to derive as explained in [Sil10]. In short, the subject reduction property for this system cannot be derived easily with a straight induction. The main reason is that the base case needs a property called Product injectivity (1.4.2). This case is detailed below.

$$\frac{\Gamma \vdash^e_{\mathscr{C}} A \equiv_\beta B : s}{\Gamma \vdash^e_{\mathscr{C}} A \preceq_{\mathscr{C}} B : s} \preceq^e_{\equiv_\beta} \qquad \frac{(s,s') \in \mathcal{C}^*_{\mathscr{C}} \qquad \Gamma \vdash^e_{\mathscr{C}} s' : s''}{\Gamma \vdash^e_{\mathscr{C}} s \preceq_{\mathscr{C}} s' : s''} \preceq^e_{\mathcal{C}^*_{\mathscr{C}}}$$

$$\frac{(s,s') \in \mathcal{C}^*_{\mathscr{C}} \qquad s' \in \mathcal{S}^\top_{\mathscr{C}}}{\Gamma \vdash^e_{\mathscr{C}} s \preceq_{\mathscr{C}} s' : s_\infty} \preceq^e_{\mathcal{C}^*_{\mathscr{C}}{}^\top}$$

$$\frac{\Gamma \vdash^e_{\mathscr{C}} A : s_1 \qquad \Gamma \vdash^e_{\mathscr{C}} B \preceq_{\mathscr{C}} B' : s_2 \qquad (s_1,s_2,s_3) \in \mathcal{R}_{\mathscr{C}}}{\Gamma \vdash^e_{\mathscr{C}} (x:A) \to B \preceq_{\mathscr{C}} (x:A) \to B' : s_3} \preceq^e_\Pi$$

$$\frac{\Gamma \vdash^e_{\mathscr{C}} A \preceq_{\mathscr{C}} B : s \qquad \Gamma \vdash^e_{\mathscr{C}} B \preceq_{\mathscr{C}} C : s'}{\Gamma \vdash^e_{\mathscr{C}} A \preceq_{\mathscr{C}} C : s'} \preceq^e_{tr}$$

Figure 3.4: Explicit CTS subtyping relation

Lemma 3.3.3 (Subject reduction) *If* $\Gamma \vdash_{\mathscr{C}} t : A$ *and* $t \hookrightarrow_\beta t'$ *then* $\Gamma \vdash_{\mathscr{C}} t' : A$

Proof *By induction on* $t \hookrightarrow_\beta t'$

◊ $\hookrightarrow_\beta$: $t = (\lambda x:B.\,t_1)\ t_2, t' = t_1\{x \leftarrow t_2\}$

	(1)	$\Gamma \vdash_{\mathscr{C}} (\lambda x:B.\,t_1)\ t_2 : A$	*Main Hypothesis*	
	(2)	$\Gamma \vdash_{\mathscr{C}} \lambda x:B.\,t_1 : (x:C) \to D$	*Inversion on application*	*1*
	(3)	$\Gamma \vdash_{\mathscr{C}} t_2 : C$		
	(4)	$D\{x \leftarrow t_2\} \preceq_{\mathscr{C}} A$		
	(5)	$\Gamma \vdash_{\mathscr{C}} D\{x \leftarrow t_2\}$ **ws**		
	(6)	$\Gamma, x:B \vdash_{\mathscr{C}} t_1 : F$	*Inversion on abstraction*	*2*
	(7)	$\Gamma \vdash_{\mathscr{C}} (x:B) \to F : s'$		
	(8)	$(x:B) \to F \preceq_{\mathscr{C}} (x:C) \to D$		
	(9)	$B \equiv_\beta C$	*Product injectivity (1.4.2)*	*8*
	(10)	$F \preceq_{\mathscr{C}} D$		
	(11)	$\Gamma \vdash_{\mathscr{C}} B : s_1$	*Inversion on product*	*7*
	(12)	$\Gamma \vdash_{\mathscr{C}} t_2 : B$	$\mathscr{C}_\preceq$	*3,11,9*
	(13)	$\Gamma \vdash_{\mathscr{C}} (x:C) \to D$ **ws**	*Well-sorted*	*2*
	(14)	$\Gamma \vdash_{\mathscr{C}} (x:C) \to D : s_3$	*Inversion on* $\mathbf{ws}_{type}$	*13*
	(15)	$\Gamma \vdash_{\mathscr{C}} D : s_4$	*Inversion on product*	*14*
	(16)	$\Gamma \vdash_{\mathscr{C}} t_1\{x \leftarrow t_2\} : F\{x \leftarrow t_2\}$	*Substitution lemma*	*6,12*
	(17)	$F\{x \leftarrow t_2\} \preceq_{\mathscr{C}} D\{x \leftarrow t_2\}$	*Lemma (1.4.3)*	*10*
	(18)	$\Gamma \vdash_{\mathscr{C}} t_1\{x \leftarrow t_2\} : D\{x \leftarrow t_2\}$	*Well-sorted subtyping*	*16,5,17*
	(19)	$\Gamma \vdash_{\mathscr{C}} A$ **ws**	*Well-sorted*	*1*
✯	(20)	$\Gamma \vdash_{\mathscr{C}} t_1\{x \leftarrow t_2\} : A$	*Well-sorted subtyping*	*18,19,4*

◊ Other cases: *Full proof can be found in [AGM92] (Chapter 5, Theorem 5.2.15).*

All the other cases for subject reduction can be transposed quite easily with an explicit typed conversion, so therefore it is really the injectivity of product which is blocking. And there is

$$\frac{\Gamma \vdash^e_{\mathscr{C}} t \equiv_\beta u : A \qquad \Gamma \vdash^e_{\mathscr{C}} A \preceq_{\mathscr{C}} B : s}{\Gamma \vdash^e_{\mathscr{C}} t \equiv_\beta u : B}\ \mathscr{C}^{\equiv_\beta}$$

$$\frac{\Gamma \vdash^e_{\mathscr{C}} M \equiv_\beta M' : (x : A) \to B \qquad \Gamma \vdash^e_{\mathscr{C}} N \equiv_\beta N' : A}{\Gamma \vdash^e_{\mathscr{C}} M\ N \equiv_\beta M'\ N' : B\{x \leftarrow N\}}\ \mathscr{C}^{\equiv_\beta}_{\text{app}}$$

$$\frac{\Gamma \vdash_{\mathscr{C}} \mathbf{wf} \qquad (s, s') \in \mathcal{A}_{\mathscr{C}}}{\Gamma \vdash^e_{\mathscr{C}} s \equiv_\beta s : s'}\ \mathscr{C}^{\equiv_\beta}_{\text{sort}}$$

$$\frac{\Gamma \vdash^e_{\mathscr{C}} A \equiv_\beta A' : s_1 \qquad \Gamma \vdash^e_{\mathscr{C}} B \equiv_\beta B' : s_2 \qquad (s_1, s_2, s_3) \in \mathcal{R}_{\mathscr{C}}}{\Gamma \vdash^e_{\mathscr{C}} (x : A) \to B \equiv_\beta (x : A') \to B' : s_3}\ \mathscr{C}^{\equiv_\beta}_{\Pi}$$

$$\frac{\begin{array}{c}\Gamma \vdash^e_{\mathscr{C}} A \equiv_\beta A' : s_1 \\ \Gamma, x : A \vdash^e_{\mathscr{C}} M : B \qquad (s_1, s_2, s_3) \in \mathcal{R}_{\mathscr{C}} \qquad \Gamma, x : A \vdash^e_{\mathscr{C}} M \equiv_\beta M' : B\end{array}}{\Gamma \vdash^e_{\mathscr{C}} \lambda x : A.\, M \equiv_\beta \lambda x : A'.\, M' : (x : A) \to B}\ \mathscr{C}^{\equiv_\beta}_{\lambda}$$

$$\frac{\Gamma \vdash^e_{\mathscr{C}} M : A}{\Gamma \vdash^e_{\mathscr{C}} M \equiv_\beta M : A}\ \mathscr{C}^{\equiv_\beta}_{\text{refl}}$$

$$\frac{\Gamma \vdash^e_{\mathscr{C}} M \equiv_\beta N : A}{\Gamma \vdash^e_{\mathscr{C}} N \equiv_\beta M : A}\ \mathscr{C}^{\equiv_\beta}_{\text{sym}}$$

$$\frac{\Gamma \vdash^e_{\mathscr{C}} M \equiv_\beta N : A \qquad \Gamma \vdash^e_{\mathscr{C}} N \equiv_\beta O : A}{\Gamma \vdash^e_{\mathscr{C}} M \equiv_\beta O : A}\ \mathscr{C}^{\equiv_\beta}_{\text{trans}}$$

$$\frac{\begin{array}{c}\Gamma \vdash^e_{\mathscr{C}} A : s_1 \\ \Gamma, x : A \vdash^e_{\mathscr{C}} B : s_2 \qquad \Gamma \vdash^e_{\mathscr{C}} N : A \qquad \Gamma, x : A \vdash^e_{\mathscr{C}} M : B \qquad (s_1, s_2, s_3) \in \mathcal{R}_{\mathscr{C}}\end{array}}{\Gamma \vdash^e_{\mathscr{C}} (\lambda x : A.\, M)\ N \equiv_\beta M\{x \leftarrow N\} : B\{x \leftarrow N\}}\ \mathscr{C}^{\equiv_\beta}_{\text{beta}}$$

Figure 3.5: Derivation rules of β

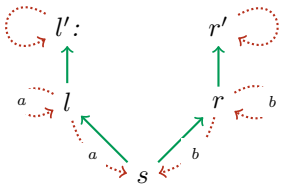
a good reason which is the transposed version of the injectivity of product in the explicit type system is false as proved below.

Lemma 3.3.4 *Following example from [SH10], we can construct a PTS which shows that there exist $\mathscr{C}$, Γ, A, B, C, D, s such that*

- $\Gamma \vdash_{\mathscr{C}} (x:A) \to B : s$
- $\Gamma \vdash_{\mathscr{C}} (x:C) \to D : s$
- $(x:A) \to B \equiv_\beta (x:C) \to D$

But there is no sort s' such that $\Gamma \vdash_{\mathscr{C}} A : s$ and $\Gamma \vdash_{\mathscr{C}} C : s$.

Proof *To falsify this statement, we use a non-functional PTS. The idea is to have two terms A and C such that $A \hookrightarrow_\beta s \hookrightarrow_\beta^* C$ but it is not possible to type A with the type of C and vice-versa. Let us define the specification $\mathscr{E}$ as:*



Then let us define the following terms:

- $A = (\lambda x:l.\, s)\ s$
- $B = s$
- $C = (\lambda x:r.\, s)\ s$
- $D = s$

Then one can derive

- $\vdash^e_{\mathscr{E}} A : l$
- $\vdash^e_{\mathscr{E}} C : r$
- $\vdash^e_{\mathscr{E}} A \to B : s$
- $\vdash^e_{\mathscr{E}} C \to D : s$
- $\vdash^e_{\mathscr{E}} A \to B \equiv_\beta C \to D : s$

But one cannot derive $\vdash^e_{\mathscr{E}} A \equiv_\beta C : l$ or $\vdash^e_{\mathscr{E}} A \equiv_\beta C : r$ because this would imply that $\vdash_{\mathscr{E}} A : r$ or $\vdash_{\mathscr{E}} C : l$ using Lemma 3.3.1 which is not possible in $\mathscr{E}$.

This lemma gives a counter-example to the *false* statement: If $\Gamma \vdash^e_{\mathscr{C}} (x:A) \to B \equiv_\beta (x:C) \to D : s$ then there exists s_1 and s_2 such that $\Gamma \vdash^e_{\mathscr{C}} A \equiv_\beta C : s_1$ and $\Gamma, x : A \vdash^e_{\mathscr{C}} B \equiv_\beta D : s_2$.

This lemma also motivates the rule $\preceq^e_{tr}$. Indeed we see that we can have $\Gamma \vdash^e_{\mathscr{C}} A \equiv_\beta B : s$ and $\Gamma \vdash^e_{\mathscr{C}} B \equiv_\beta C : s'$ having neither $s = s'$ nor $\Gamma \vdash^e_{\mathscr{C}} A \equiv_\beta C : s'$ and vice-versa. However, following the discussion initiated in Section 1.7.2, in the case where every term are well-typed, the transitivity rules $\preceq^e_{tr}$ and $\mathscr{C}^{\equiv_\beta}_{\text{trans}}$ are probably not necessary since they can be simulated with several applications of $\mathscr{C}_{\preceq}$ and $\mathscr{C}^s_{\preceq}$ since all intermediate types are proved well-sorted.

We cannot use subject reduction on the original system to derive this equivalence, because subject reduction on the original system may introduce new untyped conversion. These new conversions require to use again subject reduction, which may again introduce new untyped conversion and so on.... which leads to a circular proof. Our idea, is to use well-structured derivation trees to break this circularity. We show that in the circular argument mentioned previously, the level of the derivation tree decreases strictly between two calls to the subjection reduction lemma.

Definition 3.3.2 (EIE)
We define the equivalence between the explicit and implicit system at level n (EIE_n) as:

- $WS_n(\Gamma \vdash_{\mathscr{C}} t : A)$ *if and only if* $\Gamma \vdash^e_{\mathscr{C}} t : A$
- $WS_n(\Gamma \vdash_{\mathscr{C}} \mathbf{wf})$ *if and only if* $\Gamma \vdash^e_{\mathscr{C}} \mathbf{wf}$
- *If* $WS_n(\Gamma \vdash_{\mathscr{C}} t : A)$ *and* $t \hookrightarrow_\beta t'$ *then* $\Gamma \vdash^e_{\mathscr{C}} t \equiv_\beta t' : A$

We define EIE as for all $n \in \mathbb{N}$, EIE_n.

Using well-structured derivation trees, the idea is to prove EIE_{n+1} using subject reduction at level n. Similarly to what we have done in Section 1.7.2, we need first to introduce a lemma about the subtyping.

Lemma 3.3.5 *Assuming EIE_n, if $WS_n(\Gamma \vdash_{\mathscr{C}} A \ \mathbf{ws})$, $WS_n(\Gamma \vdash_{\mathscr{C}} B \ \mathbf{ws})$ and $A \preceq_{\mathscr{C}} B$ then there exists $\Gamma \vdash^e_{\mathscr{C}} A \preceq_{\mathscr{C}} B : s$.*

Proof *Induction will fail because of the transitivity rule. This is why we will use the subtyping relation $\preceq^{t^-}_{\mathscr{C}}$ which defines the same relation as $\preceq_{\mathscr{C}}$ (see Lemma 1.7.16). The proof is straightforward and similar to the proof of Lemma 1.7.18.*

The key lemma to prove is therefore the following

Lemma 3.3.6 *Assuming*

- EIE_n,
- $\forall \Delta, u, B, WS_{n+1}(\Delta \vdash_{\mathscr{C}} u : B) \iff \Delta \vdash^e_{\mathscr{C}} u : B$
- $\forall \Delta\, WS_{n+1}(\Delta \vdash_{\mathscr{C}} \mathbf{wf}) \iff \Delta \vdash^e_{\mathscr{C}} \mathbf{wf}$
- $WS_{n+1}(\Gamma \vdash_{\mathscr{C}} t : A)$
- $t \hookrightarrow_\beta t'$

then $\Gamma \vdash^e_{\mathscr{C}} t \equiv_\beta t' : A$.

Proof *As explained previously, only the base case is different because of the injectivity of product. Hence, we will only handle the base case here. All the other cases are straightforward.*

◊ $\hookrightarrow_\beta$: $t = (\lambda x\!:\!B.\,t_1)\ t_2, t' = t_1\,\{x \leftarrow t_2\}$

The main trick is to construct derivation trees without changing the level so that we can use injectivity of product in the CTS with untyped conversions.

	(1)	$WS_{n+1}(\Gamma \vdash_{\mathscr{C}} (\lambda x\!:\!B.\,t_1)\ t_2 : A)$	*Main Hypothesis*	
	(2)	EIE_{n+1}		
	(3)	$\forall \Delta, u, B,\ WS_{n+1}(\Delta \vdash_{\mathscr{C}} u : B) \iff \Delta \vdash^e_{\mathscr{C}} u : B$		
	(4)	$WS_{n+1}(\Gamma \vdash_{\mathscr{C}} \lambda x\!:\!B.\,t_1 : (x\!:\!C) \to D)$	*Inversion on application*	*1*
	(5)	$WS_{n+1}(\Gamma \vdash_{\mathscr{C}} t_2 : C)$		
	(6)	$D\,\{x \leftarrow t_2\} \preceq_{\mathscr{C}} A$		
	(7)	$WS_n(\Gamma \vdash_{\mathscr{C}} D\,\{x \leftarrow t_2\}\ \mathbf{ws})$		
	(8)	$WS_{n+1}(\Gamma, x : B \vdash_{\mathscr{C}} t_1 : F)$	*Inversion on abstraction*	*4*
	(9)	$WS_n(\Gamma \vdash_{\mathscr{C}} (x\!:\!B) \to F : s')$		
	(10)	$(x\!:\!B) \to F \preceq_{\mathscr{C}} (x\!:\!C) \to D$		
	(11)	$WS_n(\Gamma \vdash_{\mathscr{C}} (x\!:\!C) \to D : s)$		
	(12)	$B \equiv_\beta C$	*Product injectivity (1.4.2)*	*10*
	(13)	$F \preceq_{\mathscr{C}} D$		
	(14)	$WS_n(\Gamma \vdash_{\mathscr{C}} B : s_1)$	*Inversion on product*	*9*
	(15)	$(s_1, s_2, s_3) \in \mathcal{R}_{\mathscr{C}}$		
	(16)	$WS_n(\Gamma, x : B \vdash_{\mathscr{C}} D : s)$	*Inversion on product*	*11*
	(17)	$WS_{n+1}(\Gamma, x : B \vdash_{\mathscr{C}} t_1 : D)$	*Well-sorted subtyping*	*8,16,13*
	(18)	$\Gamma, x : B \vdash^e_{\mathscr{C}} t_1 : D$	*Typing equivalence at* $n+1$	*3,17*
	(19)	$WS_{n+1}(\Gamma \vdash_{\mathscr{C}} t_2 : B)$	*Lemma (3.1.3)*	*5,14,12*
	(20)	$\Gamma \vdash^e_{\mathscr{C}} t_2 : B$	*Typing equivalence at* $n+1$	*3,19*
	(21)	$\Gamma \vdash^e_{\mathscr{C}} B : s_1$	EIE_n	*2,14*
	(22)	$\Gamma, x : B \vdash^e_{\mathscr{C}} D : s_2$	EIE_n	*2,16*
	(23)	$\Gamma \vdash^e_{\mathscr{C}} (\lambda x\!:\!B.\,t_1)\ t_2 \equiv_\beta t_1\,\{x \leftarrow t_2\} : D\,\{x \leftarrow t_2\}$	$\mathscr{C}^{=_\beta}_{\text{beta}}$	*21,22,20,18,15*
	(24)	$\Gamma \vdash_{\mathscr{C}} A\ \mathbf{ws}$	*Well-sorted*	*1*
	(25)	$\Gamma \vdash^e_{\mathscr{C}} D\,\{x \leftarrow t_2\} \preceq_{\mathscr{C}} A : s$	*Lemma (3.3.5)*	*24,7,6*
★	(26)	$\Gamma \vdash^e_{\mathscr{C}} (\lambda x\!:\!B.\,t_1)\ t_2 \equiv_\beta t_1\,\{x \leftarrow t_2\} : A$	$\mathscr{C}^{=_\beta}$	*23,25*

Other cases*: The other cases follow by induction (see [Sil10]).*

Using the lemma above, one may prove the following induction step:

Lemma 3.3.7 *If* EIE_n *then* EIE_{n+1}.

Proof *First we prove the two equivalences*

- $WS_{n+1}(\Gamma \vdash_{\mathscr{C}} t : A) \iff \Gamma \vdash^e_{\mathscr{C}} t : A$
- $WS_{n+1}(\Gamma \vdash_{\mathscr{C}} \mathbf{wf}) \iff \Gamma \vdash^e_{\mathscr{C}} \mathbf{wf}$

By induction on the typing derivation. Using the well-structured hypothesis, we need subject reduction only at level n *to handle the subtyping cases. Finally, we conclude the equivalence proof using Lemma (3.3.6).*

Theorem 3.3.8 *For all* n*, we have if* EIE_n*. In particular we have* $WS_n(\Gamma \vdash_{\mathscr{C}} t : A)$ *then* $\Gamma \vdash^e_{\mathscr{C}} t : A$.

Proof *By induction on* n*. The base cases are trivial (every type is a sort). The induction step is handled by Lemma (3.3.7).*

The same remark can be done that for expansion postponement: Conjecture 9 implies that this equivalence is also true for all derivation trees and therefore for all specifications. However, we have not result about the opposite direction yet. One interesting idea would be to analyse a direct proof of subject reduction for semantic PTS using Siles [Sil10] results. At this time, the only proof known using is indirect since it uses subject reduction of another typing system. Such proof would give us insights on how a level of a derivation tree may grow through a substitution. We will see below, we believe that the substitution lemma is the corner-stone behind our conjecture about well-structure derivation trees.

3.4 About Well-Structured derivation trees

At this point, we know that if a tree is well-structured we can derive many properties about it. We have explored two which are:

- Expansion postponement
- The equivalence of typed and untyped conversion

However, we gave so far only trivial examples of well-structured derivations trees. We conjecture that actually every derivation tree are well-structured.

Conjecture 9 (Well-structured derivation trees) *For any CTS specification $\mathscr{C}$, every derivable derivation tree in $\mathscr{C}$ is well-structured.*

However, the truthfulness of this conjecture has many implications. In particular it would solves other conjectures considered as very difficult problems. Because the well-structured hypothesis will be used in the upcoming development of this thesis, we wish to give insights on why this property does not only apply on really simple proofs such as the one of the Simply Typed Lambda Calculus and also why we believe that this conjecture is true. In particular we have experimented these ideas on the arithmetic proofs we have used in the second part of this thesis to show that these proofs are actually well-structured.

In the next sections we discuss some attempts to derive criteria to derive automatically well-structured derivation trees.

3.4.1 Deriving well-structured derivation trees

Our goal is to annotate judgments with a level so that from any derivable judgment we can automatically construct the family $(\mathcal{L}n)_{n\in\mathbb{N}}$ as the union of subtrees derivable at level n. A naive approach is given in Figure 3.6.

We emphasize that in this system (and the ones after), levels are a property of a derivation tree and not simply a judgment annotation as shown in the example below.

Example 3.4 *In the specification $\star$, the following derivation tree for the judgment $\varnothing \vdash_\star (A:\star) \to A:\star$ is derivable at level* 1.

$$\dfrac{\dfrac{}{\vdash^0_\star \star:\star} \qquad \dfrac{\vdash^0_\star \star:\star}{A:\star \vdash^1_\star A:\star} \qquad (\star,\star,\star)\in\star}{\varnothing \vdash^1_\star (A:\star)\to A:\star}$$

Since there is no β redex one can check that all the conditions to be well-structured are satisfied. However, this does not mean that all the derivation trees for the judgment $\varnothing \vdash^1_\star$

$$\frac{}{\varnothing \vdash^n_{\mathscr{C}} \mathbf{wf}}\ \mathscr{C}^{\mathbf{wf}}_{\varnothing}$$

$$\frac{\Gamma \vdash^n_{\mathscr{C}} A : s \qquad x \notin \Gamma}{\Gamma, x : A \vdash^n_{\mathscr{C}} \mathbf{wf}}\ \mathscr{C}^{\mathbf{wf}}_{var}$$

$$\frac{\Gamma \vdash^n_{\mathscr{C}} \mathbf{wf} \qquad (x : A) \in \Gamma}{\Gamma \vdash^{n+1}_{\mathscr{C}} x : A}\ \mathscr{C}_{var}$$

$$\frac{\Gamma \vdash^n_{\mathscr{C}} \mathbf{wf} \qquad (s_1, s_2) \in \mathcal{A}}{\Gamma \vdash^n_{\mathscr{C}} s_1 : s_2}\ \mathscr{C}_{sort}$$

$$\frac{\Gamma \vdash^n_{\mathscr{C}} A : s_1 \qquad \Gamma, x : A \vdash^n_{\mathscr{C}} B : s_2 \qquad (s_1, s_2, s_3) \in \mathcal{R}}{\Gamma \vdash^n_{\mathscr{C}} (x : A) \to B : s_3}\ \mathscr{C}_{\Pi}$$

$$\frac{\Gamma, x : A \vdash^{n+1}_{\mathscr{C}} M : B \qquad \Gamma \vdash^n_{\mathscr{C}} (x : A) \to B : s}{\Gamma \vdash^{n+1}_{\mathscr{C}} \lambda x : A.\, M : (x : A) \to B}\ \mathscr{C}_{\lambda}$$

$$\frac{\Gamma \vdash^n_{\mathscr{C}} M : (x : A) \to B \qquad \Gamma \vdash^n_{\mathscr{C}} N : A}{\Gamma \vdash^n_{\mathscr{C}} M\ N : B\{x \leftarrow N\}}\ \mathscr{C}_{app}$$

$$\frac{\Gamma \vdash^{n+1}_{\mathscr{C}} M : A \qquad \Gamma \vdash^n_{\mathscr{C}} B : s \qquad A \preceq_{\mathscr{C}} B}{\Gamma \vdash^{n+1}_{\mathscr{C}} M : B}\ \mathscr{C}_{\preceq}$$

$$\frac{\Gamma \vdash^n_{\mathscr{C}} M : A \qquad A \preceq_{\mathscr{C}} s}{\Gamma \vdash^n_{\mathscr{C}} M : s}\ \mathscr{C}^{s}_{\preceq}$$

Figure 3.6: Typing rules for CTS with (wrong) levels

$(A : \star) \to A : \star$ *are derivable at level* 1. *For example, one can introduce an expansion* $\star \hookleftarrow_\beta (\lambda x : \star.\, x)\ \star$. *In that case, this new derivation tree is derivable at level* 3.

Remark 20 *In this system, the level denotes the maximum length of a chain* $\pi_0 \prec \pi_1 \prec \cdots \prec \pi_n$. *Indeed, a well-typed sort is derivable at level* 0, *a type which is typable by a sort is derivable at level* 1, *etc... This is why the type* $(\lambda x : \star.\, x)\ \star$ *is derivable at level* 2. *The type* $\star$ *is derivable at level* 0 *and thus, the type* $\star \to \star$ *is derivable at level* 1 *which is the type of* $(\lambda x : \star.\, x)\ \star$. *Also we would like to draw the reader's attention on the fact that this notion of level is independent from the specification considered. Indeed, using Theorem 2.1.2, there is one canonical embedding from every derivation tree to the specification* $\star$. *This canonical embedding preserves the shape of the derivation tree. Therefore, proving the conjecture on* $\star$ *is enough to show that the conjecture can be derived for all specifications.*

Using this system, it is easy to extract a family $(\mathcal{L}n)_{n\in\mathbb{N}}$ and to verify that this family satifies all the properties to be well-structured except $WS_{\hookrightarrow_\beta}$. This last property is harder to check

because we need to ensure that it is true for every reduction which may happen anywhere in the derivation tree. To do so, we need to prove a substitution lemma which shows that levels are *stable* by substitution. However, in this system it is not true.

Example 3.5 *One may derive* $Y : \star, X : Y \vdash^1_\star X : Y$ *and* $\vdash^2_\star (\lambda z : \star.\, \star)\ \star : \star$ *but if we substitute* $(\lambda z : \star.\, \star)\ \star$ *for* Y *we obtain the judgment* $X : (\lambda z : \star.\, \star)\ \star \vdash^3_\star X : (\lambda z : \star.\, \star)\ \star$ *which can only be derived at level* 3.

Hence if we denote $A = (\lambda z : \star.\, \star)\ \star$, *we have* $\vdash^2_\star (\lambda Y : \star.\, \lambda X : Y.\, X)\ A : A \to A$ *but* $\vdash^3_\star \lambda X : A.\, X : A \to A$.

The example above shows that levels *are not stable by substitution*! However, this does not mean that this derivation tree is not well-structured: Only, we did not construct the appropriate family $(\mathcal{L}n)_{n\in\mathbb{N}}$ with the typing system above. Coming back to our example, we only need to put the derivation tree $\vdash_{\mathscr{C}} (\lambda Y : \star.\, \lambda X : Y.\, X)\ A : A \to A$ at level 3 instead. This example also shows that the level of a type can increase through a substitution which of course causes the levels of all the subsequent derivation trees to increase.

If we assume for a moment that one never needs to apply a substitution in a type because all types are closed terms for example, then it is easy to show that level are stable by substitution and therefore, stable by β reduction.

Definition 3.4.1 (Silent substitution)
We say that a substitution $\sigma = \{x \leftarrow N\}$ *is* silent *with respect to a derivation tree* π *if whenever* $\pi' \lhd \pi$, *such that* $\dfrac{\pi'}{\Gamma \vdash_{\mathscr{C}} t : A}$, *then* $x \notin \mathrm{FV}(t)$.

Corollary 3.4.1 *If a substitution* σ *is* silent *in a derivation tree* π *then it is also silent for all derivations* π' *such that* $\pi' \prec \pi$.

Lemma 3.4.2 *A silent substitution does not change the level of a derivation: If* π *is a derivation of* $\Gamma, x : A, \Gamma' \vdash^n_{\mathscr{C}} t : B$, $N \vdash^m_{\mathscr{C}} A :$ *and* $\sigma = \{x \leftarrow N\}$ *is silent in* π *then* $\Gamma, \Gamma'\sigma \vdash^n_{\mathscr{C}} t\sigma : B\sigma$.

Definition 3.4.2 (Silent derivation tree)
A derivation tree π *is* silent *if for all* π', π_1 *such that* $\pi' \prec \pi$ *and* $\pi \hookrightarrow_\beta \pi_1$, *then the substitution generated by the* β *reduction is silent in* π'.

Theorem 3.4.3 *A* silent *derivation tree is well-structured.*

Proof *Trivial by induction using lemma 3.4.2: A substitution applied to a type of the derivation tree is always silent.*

Corollary 3.4.4 *Every derivation tree derivable in* Simply Typed Lambda Calculus *are well-structured.*

3.4.2 A variant of CTS

The type system presented in Fig 3.6 allows us to show that *silent* derivation trees are well-structured. In this section, we explore a variant of this type system which allows us to show that a larger class of derivation trees is well-structured. If we analyze our failure on Example 3.5, we see that the way we compute levels is wrong for the application rule $\mathscr{C}_{app}$! Since types are not stable by substitution, the level of $B\,\{x \leftarrow N\}$ may be higher than n! This is actually what this example shows.

A solution to overcome this issue is therefore to ensure that $B\{x \leftarrow N\}$ is well-sorted at some level, by adding the well-sorted derivation of $B\{x \leftarrow N\}$ as a premise of the rule $\mathscr{C}_{app}$. We show here why this premise can be safely added to the rule $\mathscr{C}_{app}$ without changing the expressivity of the system.

Definition 3.4.3 (Variant of CTS)
We define the typing judgment $\Gamma \vdash^a_{\mathscr{C}} t : A$ *identical to* $\Gamma \vdash_{\mathscr{C}} t : A$ *but the rule* $\mathscr{C}_{app}$ *is replaced by the one below:*

$$\frac{\Gamma \vdash^a_{\mathscr{C}} M : (x:A) \to B \qquad \Gamma \vdash^a_{\mathscr{C}} N : A \qquad \Gamma \vdash^a_{\mathscr{C}} B\{x \leftarrow N\} : s}{\Gamma \vdash^a_{\mathscr{C}} M\ N : B\{x \leftarrow N\}}\ \mathscr{C}^a_{app}$$

To prove that $\Gamma \vdash_{\mathscr{C}} t : A$ and $\Gamma \vdash^a_{\mathscr{C}} t : A$ define the same typing relation, we first need to prove the substitution lemma for this new type system. This requires other lemmas such as inversion lemmas which remain true for this new system such as the inversion lemma on products:

Lemma 3.4.5 (Inversion Π) *If* $\Gamma \vdash_{\mathscr{C}} t : (x:A) \to B$ *then* $\Gamma, x : A \vdash^a_{\mathscr{C}} B : s$.

Proof *Same proof as for usual CTS.*

The proof of the substitution lemma remains the same for every case except the application case which is handled below.

Lemma 3.4.6 (Substitution lemma) *If* $\Gamma, x : A, \Gamma' \vdash^a_{\mathscr{C}} t : B$ *and* $\Gamma \vdash^a_{\mathscr{C}} N : A$ *then* $\Gamma, \Gamma'\{x \leftarrow N\} \vdash^a_{\mathscr{C}} t\{x \leftarrow N\} : B\{x \leftarrow N\}$

Proof *By induction on the derivation of* $\Gamma, x : A, \Gamma' \vdash^a_{\mathscr{C}} t : B$. *All the cases are the same as usual CTS except the application case:*

$\Diamond$ $\mathscr{C}^a_{app}$: $t = t_1\ t_2, B = D\{y \leftarrow t_2\}, \sigma = \{x \leftarrow N\}$

	(1)	$\Gamma, x : A, \Gamma' \vdash^a_{\mathscr{C}} t_1\ t_2 : D\{y \leftarrow t_2\}$	*Main hypothesis*	
	(2)	$\Gamma \vdash^a_{\mathscr{C}} N : A$		
	(3)	$\Gamma, x : A, \Gamma' \vdash^a_{\mathscr{C}} t_1 : (y:C) \to D$	*Inversion* $\mathscr{C}^a_{app}$	*1*
	(4)	$\Gamma, x : A, \Gamma' \vdash^a_{\mathscr{C}} t_2 : C$		
	(5)	$\Gamma, x : A, \Gamma' \vdash^a_{\mathscr{C}} D\{y \leftarrow t_2\} : s$		
	(6)	$\Gamma, \Gamma'\sigma \vdash^a_{\mathscr{C}} t_1\sigma : ((y:C) \to D)\,\sigma$	*Induction hypothesis*	*3*
	(7)	$\Gamma, \Gamma'\sigma \vdash^a_{\mathscr{C}} t_2\sigma : C\sigma$	*Induction hypothesis*	*4*
	(8)	$\Gamma, \Gamma'\sigma \vdash^a_{\mathscr{C}} (D\{y \leftarrow t_2\})\,\sigma : s\sigma$	*Induction hypothesis*	*5*
	(9)	$\Gamma, \Gamma'\sigma \vdash^a_{\mathscr{C}} t_1\sigma : (y:C\sigma) \to D\sigma$	*Substitution*	*6*
	(10)	$\Gamma, \Gamma'\sigma \vdash^a_{\mathscr{C}} D\sigma\{y \leftarrow t_2\sigma\} : s$	*Substitution*	*8*
	(11)	$\Gamma, \Gamma'\sigma \vdash^a_{\mathscr{C}} t_1\sigma\ t_2\sigma : D\sigma\{y \leftarrow t_2\sigma\}$	$\mathscr{C}^a_{app}$	*8,9,10*
	(12)	$\Gamma, \Gamma'\sigma \vdash^a_{\mathscr{C}} (t_1\ t_2)\,\sigma : (D\{y \leftarrow t_2\})\,\sigma$	*Substitution*	*11*
✯	(13)	$\Gamma, \Gamma'\sigma \vdash^a_{\mathscr{C}} t\{x \leftarrow N\} : B\{x \leftarrow N\}$	*Definition of t,B and* σ	*12*

We can now conclude that the two type systems are equivalent.

Theorem 3.4.7 *We have the following equivalences:*

- $\Gamma \vdash_{\mathscr{C}} t : A \Leftrightarrow \Gamma \vdash^a_{\mathscr{C}} t : A$
- $\Gamma \vdash_{\mathscr{C}}$ **wf** $\Leftrightarrow \Gamma \vdash^a_{\mathscr{C}}$ **wf**

Proof *The right to left implication is trivial. The left to right implication is proved by induction on the derivation. All the cases are trivial except the application case.*

◊ $\mathscr{C}_{app}$: $t = t_1\ t_2, A = C\{x \leftarrow t_2\}$

	(1)	$\Gamma \vdash_{\mathscr{C}} t_1\ t_2 : C\{x \leftarrow t_2\}$	*Main hypothesis*	
	(2)	$\Gamma \vdash_{\mathscr{C}} t_1 : (x:B) \to C$	*Inversion on* $\mathscr{C}_{app}$	*1*
	(3)	$\Gamma \vdash_{\mathscr{C}} t_2 : B$		
	(4)	$\Gamma \vdash^a_{\mathscr{C}} t_1 : (x:B) \to C$	*Induction Hypothesis*	*2*
	(5)	$\Gamma \vdash^a_{\mathscr{C}} t_2 : B$	*Induction Hypothesis*	*3*
	(6)	$\Gamma, x:B \vdash^a_{\mathscr{C}} C : s$	*Inversion* Π *(3.4.5)*	*4*
	(7)	$\Gamma \vdash^a_{\mathscr{C}} C\{x \leftarrow t_2\} : s$	*Substitution lemma (3.4.6)*	*5,6*
	(8)	$\Gamma \vdash^a_{\mathscr{C}} t_1\ t_2 : C\{x \leftarrow t_2\}$	$\mathscr{C}^a_{app}$	*4,5,7*
✯	(9)	$\Gamma \vdash^a_{\mathscr{C}} t : A$	*Definition of* t *and* A	*8*

One advantage of this system is that the following theorem can be proven without the substitution lemma.

Theorem 3.4.8 *If* $\Gamma \vdash^a_{\mathscr{C}} t : A$ *then* $\Gamma \vdash^a_{\mathscr{C}} A$ **ws**.

Proof *By induction on* $\Gamma \vdash^a_{\mathscr{C}} t : A$. *All the cases are trivial.*

3.4.3 The next level

In this section, we show how this variant of CTS can be useful to derive new well-structured derivation trees. To do so, we replace the application rule $\mathscr{C}_{app}$ in Fig. 3.6 by the one below:

$$\frac{\Gamma \vdash^{n+1}_{\mathscr{C}} M : (x:A) \to B \qquad \Gamma \vdash^{n+1}_{\mathscr{C}} N : A \qquad \Gamma \vdash^{n}_{\mathscr{C}} B\{x \leftarrow N\} : s}{\Gamma \vdash^{n+1}_{\mathscr{C}} M\ N : B\{x \leftarrow N\}}\ \mathscr{C}_{app}$$

Example 3.6 *Going back to Example 3.5, one may show that the canonical derivation tree proving the judgment* $(\lambda Y:\star.\lambda X:Y.X)\ A \vdash^3_{\mathscr{C}} A \to A :$ *is derivable at level* 3 *in this new system. It is not really hard to show that this level is stable by* β *reduction as argued before. Hence, this new type system derives automatically the correct level allowing us to prove that this derivation tree is well-structured.*

However, this system does not allows us to prove that every derivation trees are well-structured, here is a counterexample.

Example 3.7 *There is a derivation of* $\vdash^3_\star \lambda A:\star.\lambda x:(\lambda y:\star.\star)\ A.\star : \star \to \star \to \star$. *Assuming that there is a derivation of* $\vdash^{1000}_\star N : \star$ *then one can derive* $\vdash^{1000}_\star (\lambda A:\star.\lambda x:(\lambda y:\star.\star)\ A.\star)\ N : \star \to \star$. *However, one cannot derive* $\vdash^{1000}_\star \lambda x:(\lambda y:\star.\star)\ N.\star : \star \to \star$, *it can only be derived at level* 1001 *because* $(\lambda y:\star.\star)\ N$ *will be derivable at level* 1000.

To characterize derivation trees that this new type system shows are well-structured, we first need to understand why it fails on the example above. If we take our application rule $\mathscr{C}^a_{app}$ with $f : A \to B$ and $a : A$, and assume that the substitution lemma may increase the level of a type, then in particular it can increase the level of a type A. However, because this is an elimination rule, this type disappears in the conclusion. What we would like to show, is that through the substitution lemma, we keep the property that if $\Gamma \vdash^n_{\mathscr{C}} A$ **ws** then $\Gamma \vdash^{n+1}_{\mathscr{C}} t : A$. Hence, the idea would be to prove a stronger substitution lemma which is:

Lemma 3.4.9 *If* $\Gamma, x : A, \Gamma' \vdash^n_{\mathscr{C}} t : B$, $\Gamma \vdash^m_{\mathscr{C}} N : A$ *and* $\Gamma \vdash^o_{\mathscr{C}} B\{x \leftarrow N\}$ **ws** *then we have* $\Gamma, \Gamma'\{x \leftarrow N\} \vdash^{max(n,m,o+1)}_{\mathscr{C}} t\{x \leftarrow N\} : B\{x \leftarrow N\}$.

This lemma is obviously wrong because of Example 3.7. However, notice that if we transpose this lemma for CTS by removing level annotations, this lemma is admissible in CTS and provable trivially using the classical substitution lemma. What we can do instead is to fix the statement of this lemma so that it becomes true.

Definition 3.4.4 (Whispering derivation tree)

A derivation tree $\dfrac{\pi}{\Gamma \vdash_{\mathscr{C}} t : B}$ *is said* whispering *if for every rules* $\mathscr{C}_{app}$, $\mathscr{C}_{\preceq}$ *and* $\mathscr{C}^s_{\preceq}$ *we have the following property:*

- *For every substitution* σ *and* $\pi' \prec \pi$, *then* $\Gamma \vdash^n_{\mathscr{C}} B\sigma$ **ws** *implies* π' *derivable at level* n

One may check that the derivation tree in Example 3.7 is not *whispering*.

Lemma 3.4.10 *If* $\Gamma, x : A, \Gamma' \vdash^n_{\mathscr{C}} t : B$ *is a whispering derivation,* $\Gamma \vdash^m_{\mathscr{C}} N : A$ *and* $\Gamma \vdash^o_{\mathscr{C}} B\{x \leftarrow N\}$ **ws** *then we have* $\Gamma, \Gamma'\{x \leftarrow N\} \vdash^{max(n,m,o+1)}_{\mathscr{C}} t\{x \leftarrow N\} : B\{x \leftarrow N\}$.

Proof *All the cases except* $\mathscr{C}_{app}$, $\mathscr{C}_{\preceq}$ *and* $\mathscr{C}^s_{\preceq}$ *are trivial. For these three rules, this is a a direct consequence of the definition.*

Theorem 3.4.11 *Any* whispering *derivation is well-structured.*

Proof *We need to ensure that if* $\pi \hookrightarrow_\beta \pi'$ *and* π *is derivable at level* n, *then so is* π'. *This can be done by induction on* $\pi \hookrightarrow_\beta \pi'$. *The base case is handled by Lemma 3.4.10.*

This new way to compute the family $(\mathcal{L}n)_{n\in\mathbb{N}}$ is really interesting in practice. Indeed, we were able to check empirically that all the proofs we have manipulated in the second part of this thesis were well-structured according to this definition of levels. This is done in an implementation dklevels. This implementation simply takes a proof (in Dedukti), assign levels to this proof according to the system above and checks whether levels are stable by β-reduction. It would be easier to check whether a derivation tree is *whispering* but we did not find a decidable criterion for that.[1]

3.4.4 Loud CTS

This section is exploratory but we think it is interesting to present this system because it may give some deeper insights about levels and where the difficulties are to show that all derivation trees are indeed well-structured.

If we look at again the Lemma 3.4.9, we can observe that this lemma cannot be proved by a straight induction on the derivation tree. Indeed, for the application case $\mathscr{C}_{app}$, the hypothesis for the type A is missing. This happens also for subtyping rules $\mathscr{C}_{\preceq}$ and $\mathscr{C}^s_{\preceq}$. We propose here a variant of CTS which remembers the intermediate types. In fact just remembering the type is not enough and this is why we need to remember the whole judgment. The type system for this new CTS is presented in Fig 3.7.

Here are the main three differences:

[1] An interesting path would be to check the inclusion of free variables ($\mathrm{FV}(A) \subseteq \mathrm{FV}(B)$) instead of substitutions for the definition of whispering derivation tree.

$$\frac{(s_1, s_2) \in \mathcal{A}}{\varnothing; \varnothing \vdash^n_{\mathscr{C}} s_1 : s_2}\ \mathscr{C}_{sort}$$

$$\frac{\Gamma; \Sigma \vdash^n_{\mathscr{C}} A : s \qquad x \notin \Gamma \cup \Sigma}{\Gamma, x : A; \Sigma \vdash^{n+1}_{\mathscr{C}} x : A}\ \mathscr{C}_{var}$$

$$\frac{\Gamma; \Sigma \vdash^n_{\mathscr{C}} t : A \qquad \Gamma; \Sigma \vdash^n_{\mathscr{C}} B : s \qquad x \notin \Gamma \cup \Sigma}{\Gamma, x : B; \Sigma \cup [\Gamma; \Sigma \vdash^n_{\mathscr{C}} B : s] \vdash^n_{\mathscr{C}} t : A}\ \mathscr{C}_{weak}$$

$$\frac{\Gamma; \Sigma \vdash^n_{\mathscr{C}} t : A \qquad \Delta; \Xi \vdash^n_{\mathscr{C}} B : s}{\Gamma; \Sigma \cup [\Delta; \Xi \vdash^n_{\mathscr{C}} B : s] \vdash^n_{\mathscr{C}} t : A}\ \mathscr{C}_{weaksig}$$

$$\frac{\Gamma; \Sigma \vdash^n_{\mathscr{C}} A : s_1 \qquad \Gamma, x : A; \Sigma \vdash^n_{\mathscr{C}} B : s_2 \qquad (s_1, s_2, s_3) \in \mathcal{R}}{\Gamma; \Sigma \vdash^n_{\mathscr{C}} (x : A) \to B : s_3}\ \mathscr{C}_{\Pi}$$

$$\frac{\Gamma, x : A; \Sigma \vdash^{n+1}_{\mathscr{C}} M : B \qquad \Gamma; \Sigma \vdash^n_{\mathscr{C}} (x : A) \to B : s}{\Gamma; \Sigma \cup [\Gamma; \Sigma \vdash^n_{\mathscr{C}} (x : A) \to B : s] \vdash^{n+1}_{\mathscr{C}} \lambda x : A.\, M : (x : A) \to B}\ \mathscr{C}_{\lambda}$$

$$\frac{\Gamma; \Sigma \vdash^{n+1}_{\mathscr{C}} M : (x : A) \to B \qquad \Gamma; \Sigma \vdash^{n+1}_{\mathscr{C}} N : A \qquad \Gamma; \Sigma \{x \leftarrow N\} \vdash^n_{\mathscr{C}} B \{x \leftarrow N\} : s}{\Gamma; \Sigma \cup [\Gamma; \Sigma \{x \leftarrow N\} \vdash^n_{\mathscr{C}} B \{x \leftarrow N\} : s] \vdash^{n+1}_{\mathscr{C}} M\ N : B \{x \leftarrow N\}}\ \mathscr{C}_{app}$$

$$\frac{\Gamma; \Sigma \vdash^{n+1}_{\mathscr{C}} M : A \qquad \Gamma; \Sigma \vdash^n_{\mathscr{C}} B : s \qquad A \preceq_{\mathscr{C}} B}{\Gamma; \Sigma \cup [\Gamma; \Sigma \vdash^n_{\mathscr{C}} B : s] \vdash^{n+1}_{\mathscr{C}} M : B}\ \mathscr{C}_{\preceq}$$

Figure 3.7: Typing rules for loud CTS

- We use only one judgment $\Gamma; \Sigma \vdash_{\mathscr{C}} t : A$ (and drop the well-formed judgment). We don't think this is mandatory, but it makes proofs easier.
- We use a weakening rule for typing contexts ($\mathscr{C}_{weak}$)
- We remember the type information lost in the rule $\mathscr{C}_{app}$, $\mathscr{C}_{\preceq}$ and $\mathscr{C}^s_{\preceq}$ by introducing a typing context Σ in judgments
- Σ is a set of judgments.

We leave for futur work meta-theoretical proofs for this system. The central idea behind this new system is to prove the following substitution lemma:

Conjecture 10 *If*

- $\Gamma, x : A, \Gamma'; \Sigma \vdash^n_{\mathscr{C}} t : B$
- $\Gamma; \Xi \vdash^m_{\mathscr{C}} N : A$
- $\Gamma, \Gamma' \{x \leftarrow N\}; \Xi \vdash^o_{\mathscr{C}} B \{x \leftarrow N\} : s$

- $\Sigma\{x \leftarrow N\} \subseteq \Xi$

Then

$$\Gamma, \Gamma'\{x \leftarrow N\}; \Xi \vdash_{\mathscr{C}}^{\max(n,m,o+1)} t\{x \leftarrow N\} : B\{x \leftarrow N\}$$

From this substitution lemma, we want to prove the subject reduction lemma:

Conjecture 11 *If* $\Gamma; \Sigma \vdash_{\mathscr{C}}^{n} t : A$ *and* $t \hookrightarrow_{\beta} t'$ *then there* $\Gamma; \Sigma \vdash_{\mathscr{C}}^{n} t' : A$.

This lemma is a consequence of the substitution lemma as above.

Example 3.8 *Going back to Example 3.7, We would like to express judgment*

$$\vdash_{\star}^{3} \lambda A : \star.\, \lambda x : (\lambda y : \star.\, \star)\ A.\star : \star \to \star \to \star$$

in the new system. In this judgment, the application rule is used only once, and there is no need to use subtyping. Hence the typing context can be reduced to one element. Let us denote $\Sigma = \{A : \star; \emptyset \vdash_{\mathscr{C}}^{1} \star \to \star : \star\}$. *We can derive* $\emptyset; \Sigma \vdash_{\star}^{3} \lambda A : \star.\, \lambda x : (\lambda y : \star.\, \star)\ A.\star : \star \to \star \to \star$. *Now assume that we are able to derive a term* $\emptyset; \Theta \vdash_{\star}^{1000} N : \star$. *To be able to construct the application, we need to provide a derivation of* $\emptyset; \Sigma\{A \leftarrow N\} \cup \Theta \vdash_{\star}^{n} \star : \star$ *for some* n[2]. *At that point, our example suggests that* n *should be* 1000, *but this is not mandatory, it could be any number larger than* 1, *it depends on how the term* N *is derived.*

Gathering all this, we may derive in this new system (with the rule $\mathscr{C}_{app}$*)*

$$\emptyset; \Sigma \cup \Theta \cup \Xi \vdash_{\star}^{\max(3,\max(1000,n+1))} (\lambda A : \star.\, \lambda x : (\lambda y : \star.\, \star)\ A.\star)\ N : \star \to \star$$

where Ξ *is a singleton set containing the judgment*

$$\emptyset; \Sigma \cup \Theta \vdash_{\star}^{\max(2,n)} ((\lambda y : \star.\, \star)\ A) \to \star : \star$$

.

If we take $n = 1000$ *we see that this system derives our original judgment at level* 1001 *which is what we expected.*

With the amount of information stored in the judgment, this system is indeed very loud!

In this last example We tried to hint at why we hope that this new system may derive well-structured derivation trees for any CTS derivation trees. However, considering that the substitution lemma is true, the question remains of whether the type system we have defined is equivalent to the classical system. Clearly there is an embedding from this new type system to the old one, but what about the opposite? We think this is also true because every judgment added into Σ is already derivable. However, to be able to do the same trick as for Theorem 3.4.7 we need to prove another substitution lemma which forgets the third hypothesis, and hence is similar to the classical substitution lemma. This other substitution lemma is weaker in the sense that the level may increase through a substitution. This is not an issue since it is used to construct the third premise in the application rule $\mathscr{C}_{app}$.

[2] The system should ensures that $\Theta\{A \leftarrow N\} = \Theta$

3.5 Future Work

About well-structured derivation trees We have shown that levels give a decreasing argument to solve some famous conjectures on CTS when the derivation tree is well-structured. As argued in Section 3.4, we have good reasons to believe that all CTS derivation tree are well-structured. It would be interesting to investigate more classes of CTS to show whether the derivable judgments are well-structured, in particular for terminating CTS. An attempt to prove expansion postponement for terminating systems has already been done in [Pol98] for example but the proof was wrong because of a subtle mistake.

We are aware that the path we have developed here may be wrong. In particular it is not clear whether the strong substitution lemma for loud CTS is true. Moreover, because this conjecture has many consequences we would be more confident with a proof formalized in a proof assistant. Also, it would be interesting to find a (fast) decidable criterion to check if a CTS derivation tree is whispering. This way it would be easier to ensure whether a derivation tree is well-structured without assuming our conjecture.

Well-structured derivation trees for the meta-theory of $\lambda\Pi$-calculus modulo theory Our definition of well-structured derivation trees could be adapted to the meta-theory of $\lambda\Pi$-calculus modulo theory. In particular we will mention well-structured derivation trees in Chapter 8 because it could help to solve a famous circular argument between confluence, termination and product injectivity.

Levels for $\eta\beta$ confluence in CTS In [Geu92], Herman Geuvers gives a proof that the $\beta\eta$ reductions are confluent for a large class of PTS specification. It would be interesting to see whether levels give another way to prove this famous result and see whether this could be extended for CTS.

Chapter 4

Bi-directional CTS

The type system of CTS as presented in Chapter 1 is not syntax directed, in particular it is not clear by looking at a judgment where subtyping is used. The purpose of bi-directional CTS developed in this chapter is to make explicit the use of subtyping in a judgment. This idea of bi-directional CTS is a reformulation of minimal CTS developed by Ali Assaf [Ass15b]. In bi-directional CTS, the typing judgment $\Gamma \vdash_{\mathscr{C}} t : A$ is split in two judgments: An *inference* judgment $\Gamma \vdash_{\mathscr{C}} t \Rightarrow A$ and a *checking* judgment $\Gamma \vdash_{\mathscr{C}} t \Leftarrow A$. Only checking judgments are allowed to use subtyping. In this system, subtyping is used only at the end of a proof or during an application. However, in contrast to usual bi-directional type systems [PT00], the inference judgment $\Gamma \vdash_{\mathscr{C}} t \Rightarrow A$ is not a function of Γ and t because we still have a conversion rule in the typing system. We keep this conversion rule for simplicity. Traditionally, bi-directional typing system were introduced to control conversion, but in this chapter we use them to control subtyping. At the end of this chapter we conjecture that the type system could be refined to get a syntax-directed type system.

Bi-directional CTS do not behave well for every specification. Indeed, for some of them, it is not possible to restrain subtyping for applications (see Example 4.1). This is why we define a class of CTS called *normal CTS* for which we prove in Theorem 4.3.9 an equivalence between bi-directional CTS and CTS for well-structured derivation trees (introduced in Chapter 3). The class of normal CTS is a large class of CTS which contains all the CTS specifications used for concrete proof systems.

4.1 Presentation of bi-directional CTS

Embedding CTS into the λΠ-calculus modulo theory requires to use an explicit *cast* operator because subtyping violates the type uniqueness property of the λΠ-calculus modulo theory. This property which is also true for PTS (Theorem 1.7.12) expresses that if a term is typable by two types, the these two types are convertible. This property is no longer true in a CTS where a term t can inhabit a sort s and a sort s' where s is a subtype of s'. Hence, translating the CTS judgment $\Gamma \vdash_{\mathscr{C}} t : A$ to the λΠ-calculus modulo theory requires to know when a cast is needed. However, this information only appears in the derivation tree and not in the judgment directly. Hence, to express our translation to the λΠ-calculus modulo theory, there are several solutions. Two of them are:

- Expressing the translation from CTS to the λΠ-calculus modulo theory as a function from CTS derivation trees to judgments of the λΠ-calculus modulo theory.
- Introducing a new type system of CTS to reflect subtyping on the judgment itself

The first solution has the advantage of keeping the same type system to express the translation to the λΠ-CALCULUS MODULO THEORY. However, this complexifies the soundness proof. The reason is that the image by the encoding function is a term and the proof requires that the image of two derivation trees relate in a particular way in the λΠ-CALCULUS MODULO THEORY. This way, we follow Ali Assaf's work [Ass15b].

4.1.1 Typing system for bi-directional CTS

Definition 4.1.1 (Bi-directional typing of CTS)
The bi-directional type system of CTS is defined in Figure. 4.1. It introduces two new judgments:

- $\Gamma \vdash_{\mathscr{C}} t \Rightarrow A$: *It should be read as the type A is* inferred *from the term t in the typing context* Γ
- $\Gamma \vdash_{\mathscr{C}} t \Leftarrow A$: *It should be read as the type A is* checked *against the term t in the typing context* Γ

Subtyping is used only when a term t is checked against a type A. This idea comes from bi-directional typing [PT00] and has been used by Assaf [Ass15b] to make a first translation from CTS to the λΠ-CALCULUS MODULO THEORY. One difference to note with the literature however, is that usually, the judgment $\Gamma \vdash_{\mathscr{C}} t \Rightarrow A$ can be seen as function of Γ and t while here, it is not the case because of the rules $\mathscr{C}^{\Rightarrow}_{\equiv}$ and $\mathscr{C}^{\Rightarrow}_{\equiv_s}$.

Remark 21 *In Ali Assaf's work, bi-directional was called* minimal *CTS which was related to a semantic property of the specification. We prefer to use the bi-directional terminology because we think that this system is closer to the syntax and the works initiated by Pierce [PT00].*

The goal of this chapter is to prove the equivalence between these two systems: $\Gamma \vdash_{\mathscr{C}} t : A \Leftrightarrow \Gamma \vdash_{\mathscr{C}} t \Leftarrow A$. The right-to-left implication is just an induction because the typing system of bi-directional CTS refines the typing system of CTS. Hence, the difficult implication is the other one.

A bug in subject reduction's proof for bi-directional CTS: The substitution lemma for bi-directional CTS cannot be proven easily. An example An example is the abstraction case ($\mathscr{C}^{\Rightarrow}_{\lambda}$). The goal is to prove the following statement:

- given that the judgments $\Gamma \vdash_{\mathscr{C}} N \Leftarrow A$ and $\Gamma, x : A, \Gamma' \vdash_{\mathscr{C}} \lambda x : B.\, t \Rightarrow (x : B) \to C$ are derivable. For readability, we denote $\sigma = \{x \leftarrow N\}$.
- by induction hypothesis, we have $\Gamma, x : A, \Gamma', y : B \vdash_{\mathscr{C}} t \Rightarrow C \Rightarrow \Gamma, \Gamma'\sigma, y : B\sigma \vdash_{\mathscr{C}} t\sigma \Leftarrow C\sigma$
- by induction hypothesis, we have $\Gamma, x : A, \Gamma' \vdash_{\mathscr{C}} (y : B) \to C \Rightarrow s \Rightarrow \Gamma, \Gamma'\sigma \vdash_{\mathscr{C}} (y : B\sigma) \to C\sigma \Leftarrow s$
- we want to prove that $\Gamma, \Gamma' \{x \leftarrow N\} \vdash_{\mathscr{C}} \lambda x : B\sigma.\, t\sigma \Leftarrow (x : B\sigma) \to C\sigma$ is also derivable

From the first induction hypothesis we have $\Gamma, \Gamma'\sigma, y : B\sigma \vdash_{\mathscr{C}} t\sigma \Leftarrow C\sigma$. By inversion on the typing system, we can deduce that $\Gamma, \Gamma'\sigma, y : B\sigma \vdash_{\mathscr{C}} t\sigma \Rightarrow C'$ with $C' \preceq_{\mathscr{C}} C\sigma$. From the second induction hypothesis we have $\Gamma, \Gamma'\sigma \vdash_{\mathscr{C}} (x : B\sigma) \to C\sigma \Leftarrow s$. By inversion we also have $\Gamma, \Gamma'\sigma \vdash_{\mathscr{C}} (x : B\sigma) \to C\sigma \Leftarrow D$ where $D \preceq_{\mathscr{C}} s$.

Let us assume that $C' \equiv_\beta C\sigma$. To conclude this case, we want to prove that if $\Gamma, \Gamma'\sigma \vdash_{\mathscr{C}} (x : B\sigma) \to C\sigma \Rightarrow D$ then there exists s' such that $\Gamma, \Gamma'\sigma \vdash_{\mathscr{C}} (x : B\sigma) \to C' \Rightarrow s'$. To fix this

$$\frac{}{\varnothing \vdash_{\mathscr{C}^{\Rightarrow}} \mathbf{wf}}\ \mathscr{C}^{\Rightarrow \mathbf{wf}}_{\varnothing}$$

$$\frac{\Gamma \vdash_{\mathscr{C}} A \Rightarrow s \qquad x \notin \Gamma}{\Gamma, x : A \vdash_{\mathscr{C}^{\Rightarrow}} \mathbf{wf}}\ \mathscr{C}^{\Rightarrow \mathbf{wf}}_{var}$$

$$\frac{\Gamma \vdash_{\mathscr{C}^{\Rightarrow}} \mathbf{wf} \qquad (x : A) \in \Gamma}{\Gamma \vdash_{\mathscr{C}} x \Rightarrow A}\ \mathscr{C}^{\Rightarrow}_{var}$$

$$\frac{\Gamma \vdash_{\mathscr{C}^{\Rightarrow}} \mathbf{wf} \qquad (s_1, s_2) \in \mathcal{A}}{\Gamma \vdash_{\mathscr{C}} s_1 \Rightarrow s_2}\ \mathscr{C}^{\Rightarrow}_{sort}$$

$$\frac{\Gamma \vdash_{\mathscr{C}} A \Rightarrow s_1 \qquad \Gamma, x : A \vdash_{\mathscr{C}} B \Rightarrow s_2 \qquad (s_1, s_2, s_3) \in \mathcal{R}}{\Gamma \vdash_{\mathscr{C}} (x : A) \to B \Rightarrow s_3}\ \mathscr{C}^{\Rightarrow}_{\Pi}$$

$$\frac{\Gamma, x : A \vdash_{\mathscr{C}} M \Rightarrow B \qquad \Gamma \vdash_{\mathscr{C}} (x : A) \to B \Rightarrow s}{\Gamma \vdash_{\mathscr{C}} \lambda x : A.\, M \Rightarrow (x : A) \to B}\ \mathscr{C}^{\Rightarrow}_{\lambda}$$

$$\frac{\Gamma \vdash_{\mathscr{C}} M \Rightarrow (x : A) \to B \qquad \Gamma \vdash_{\mathscr{C}} N \Leftarrow A}{\Gamma \vdash_{\mathscr{C}} M\ N \Rightarrow B\,\{x \leftarrow N\}}\ \mathscr{C}^{\Rightarrow}_{app}$$

$$\frac{\Gamma \vdash_{\mathscr{C}} M \Rightarrow A \qquad \Gamma \vdash_{\mathscr{C}} B \Rightarrow s \qquad A \equiv_{\beta} B}{\Gamma \vdash_{\mathscr{C}} M \Rightarrow B}\ \mathscr{C}^{\Rightarrow}_{\equiv}$$

$$\frac{\Gamma \vdash_{\mathscr{C}} M \Rightarrow A \qquad A \equiv_{\beta} s}{\Gamma \vdash_{\mathscr{C}} M \Rightarrow s}\ \mathscr{C}^{\Rightarrow}_{\equiv_s}$$

$$\frac{\Gamma \vdash_{\mathscr{C}} M \Rightarrow A \qquad \Gamma \vdash_{\mathscr{C}} B \Rightarrow s \qquad A \preceq_{\mathscr{C}} B}{\Gamma \vdash_{\mathscr{C}} M \Leftarrow B}\ \mathscr{C}^{\Leftarrow}_{\preceq}$$

$$\frac{\Gamma \vdash_{\mathscr{C}} M \Rightarrow A \qquad A \preceq_{\mathscr{C}} s}{\Gamma \vdash_{\mathscr{C}} M \Leftarrow s}\ \mathscr{C}^{\Leftarrow}_{\preceq_s}$$

Figure 4.1: Typing rules for bi-directional CTS

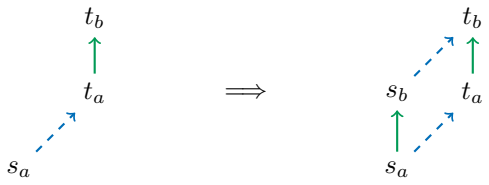


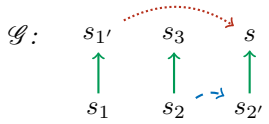
Figure 4.2: Axiom condition

problem we need to prove subject reduction for this bi-directional system. However, proving subject reduction requires the substitution lemma which itself requires subject reduction and so on...[1] This is why in our proof, we will use well-structured derivation trees (Definition 3.1.2)[2].

4.2 Normal CTS

In this section we introduce a class of CTS specification called *normal CTS*. The main reason to introduce this class, is that for some specifications, subtyping cannot be pushed to applications. Normal CTS restrains the specification with two conditions on the specification which are given in Definition 4.2.1. We show with the example below the necessity of these two conditions.

Example 4.1 *Given the specification $\mathscr{G}$ pictured by the following graph:*



In this specification, the judgment $Y : s_2 \vdash_{\mathscr{G}} \lambda x : s_1 . Y : s_1 \to s_{2'}$ is derivable. However, it is not possible to derive the judgment $Y : s_2 \vdash_{\mathscr{G}} \lambda x : s_1 . Y \Leftarrow s_1 \to s_{2'}$. The reason is because in this specification, subtyping needs to be used on the sort s_2 and cannot be postponed after the abstraction.

To avoid such pathological issue, we need to identify a class of CTS specification for which the equivalence is true. From the example above, we observe that a first restriction is to avoid top-sorts which are subtype of other sorts which may have a type. Indeed, without this condition, one needs to use subtyping to give a type to a top-sort which means that subtyping could not be pushed until an application. In the example below, this is the case of s_2. s_2 is a subtype of $s_{2'}$ and $s_{2'}$ has a type which is s. Hence our first condition to define this class is the following one: For all sorts s_a, t_a, t_b such that $(s_b, t_b) \in \mathcal{C}$ and $(t_a, t_b) \in \mathcal{A}$ then there exists s_b such that $(s_a, s_b) \in \mathcal{A}$ and $(s_b, t_b) \in \mathcal{C}^*_{\mathscr{G}}$. This condition is pictured in Figure 4.2:

Example 4.2 *Enriching the specification given in Example 4.1 we fulfill this new condition in the following specification*

[1] It appears that this bug is already present in Ali Assaf's PhD thesis [Ass15b] in Lemma 8.4.13.

[2] Another way is to restrain the class of specification for which we prove the equivalence. As usual a good candidate would be semi-full CTS (Definition 1.3.8) or full CTS (Definition 1.3.9).

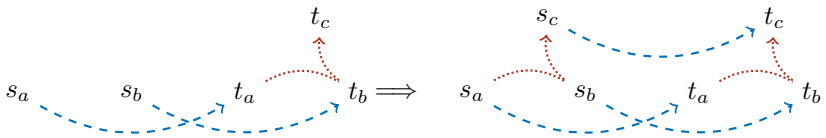
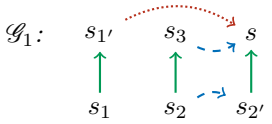


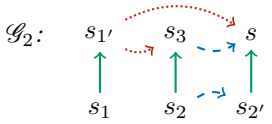
Figure 4.3: Rule condition



However, we observe that this condition is not sufficient to derive the judgment $Y : s_2 \vdash_{\mathscr{G}_1} \lambda x : s_1 . Y \Leftarrow s_1 \to s_{2'}$.

To push back the subtyping under the abstraction rule we need another condition. In $\mathscr{G}_1$, we need one of the following products $(s_{1'}, s_3, s_3)$ or (s_1, s_3, s) to derive this judgment. This leads us to a second condition which is: For all sort s_a, s_b, t_a, t_b, t_c, if $(t_a, t_b, t_c) \in \mathcal{R}_{\mathscr{C}}$, $(s_a, t_a) \in \mathcal{C}^*_{\mathscr{C}}$, $(s_b, t_b) \in \mathcal{C}^*_{\mathscr{C}}$ then there exists s_c such that $(s_a, s_b, s_c) \in \mathcal{R}_{\mathscr{C}}$ and $(s_c, t_c) \in \mathcal{C}^*_{\mathscr{C}}$. This condition is summed up in Figure 4.3:

Example 4.3 *Enriching our specification given in Example 4.1 to fulfill these two conditions give the following specification*



We observe that in this new specification, the judgment $Y : s_2 \vdash_{\mathscr{G}_2} \lambda x : s_1 . Y \Leftarrow s_1 \to s_{2'}$ *is derivable.*

The two conditions we have expressed above are enough to derive an equivalence between bi-directional CTS and CTS for well-structured derivation trees.

Definition 4.2.1 (CTS in normal form)
A CTS is said in normal form[3] *if it satisfies the following conditions:*

$$\forall (t_a, t_b) \in \mathcal{A}_{\mathscr{C}}, (s_a, t_a) \in \mathcal{C}^*_{\mathscr{C}}, \exists s_b, (s_a, s_b) \in \mathcal{A}_{\mathscr{C}} \wedge (s_b, t_b) \in \mathcal{C}^*_{\mathscr{C}} \qquad (\mathrm{NF}_{\mathcal{A}})$$

$$\forall (t_1, t_2, t_3) \in \mathcal{R}_{\mathscr{C}}, (s_a, t_a) \in \mathcal{C}^*_{\mathscr{C}}, (s_b, t_b) \in \mathcal{C}^*_{\mathscr{C}}, \exists s_c, (s_a, s_b, s_c) \in \mathcal{R}_{\mathscr{C}} \wedge (s_c, t_c) \in \mathcal{C}^*_{\mathscr{C}} \qquad (\mathrm{NF}_{\mathcal{R}})$$

This conditions are presented in Figure 4.2 and Figure 4.3.

We conjecture that any CTS is weakly CTS equivalent (Definition 2.1.12) to a CTS in normal form, which should follow from the theorems in Chapter 2.

Conjecture 12 (Equivalence between CTS and CTS in normal form) *Every CTS is equivalent to a CTS in normal form.*

[3] I think this is related to natural conditions we find in category theory.

Sketch of proof:

1. One can show that for any CTS $\mathscr{C}$, there is exists a CTS equivalent $\mathscr{C}$' which satisfies property $\mathrm{NF}_{\mathcal{R}}$ (using Theorem 2.1.4).

2. We use Corollary 2.2.16 to generate a weak-equivalent CTS specification which is top-sort regular.

3. We use Theorem 2.2.20 to generate a CTS which satifies property $\mathrm{NF}_{\mathcal{A}}$.

This last step may break property $\mathrm{NF}_{\mathcal{R}}$. For this reason, we believe that by iterating steps 1, 2 and 3 should produce a fixpoint and generate a CTS specification which is in normal form and weakly equivalent to the original specification.

4.3 Equivalence proof

Our proof relies on well-structured derivation trees. Hence, as we did in Chapter 3, we will prove one direction of the equivalence by induction on the level.

Definition 4.3.1 (EBI)
We define the equivalence between CTS and bi-directional CTS at level n (EBI_n) as:

- $WS_n(\Gamma \vdash_{\mathscr{C}} t : A)$ *then* $\Gamma \vdash_{\mathscr{C}} t \Leftarrow A$
- $WS_n(\Gamma \vdash_{\mathscr{C}} \mathbf{wf})$ *then* $\Gamma \vdash_{\mathscr{C}\Rightarrow} \mathbf{wf}$
- $\Gamma \vdash_{\mathscr{C}} t \Leftarrow A$ *then* $\Gamma \vdash_{\mathscr{C}} t : A$
- $\Gamma \vdash_{\mathscr{C}\Rightarrow} \mathbf{wf}$ *then* $\Gamma \vdash_{\mathscr{C}} \mathbf{wf}$

We define EBI as for all $n \in \mathbb{N}$, EBI_n.

The last two statements can be proved with a structural induction.

Lemma 4.3.1 (Bi-directional typing soudness) *The following judgments hold for every specification $\mathscr{C}$:*

- *If* $\Gamma \vdash_{\mathscr{C}} t \Rightarrow A$ *then* $\Gamma \vdash_{\mathscr{C}} t : A$
- *If* $\Gamma \vdash_{\mathscr{C}} t \Leftarrow A$ *then* $\Gamma \vdash_{\mathscr{C}} t : A$
- *If* $\Gamma \vdash_{\mathscr{C}\Rightarrow} \mathbf{wf}$ *then* $\Gamma \vdash_{\mathscr{C}} \mathbf{wf}$

Proof *By induction on the typing derivation. All the cases are trivial since bi-directional type system is a restriction of CTS type system.*

The other implication is harder to prove. We restate here a bit of meta-theory for bi-directional CTS.

4.3.1 Some meta-theory for bi-directional CTS

Lemma 4.3.2 (check-to-infer) *If* $\Gamma \vdash_{\mathscr{C}} t \Leftarrow C$ *then* $\exists A$ *such that* $\Gamma \vdash_{\mathscr{C}} t \Rightarrow A$ *and* $A \preceq_{\mathscr{C}} C$.

Proof *The last rule for the derivation of* $\Gamma \vdash_{\mathscr{C}} t \Leftarrow C$ *is either* $\mathscr{C}^{\Leftarrow}_{\preceq}$ *or* $\mathscr{C}^{\Leftarrow}_{\preceq_s}$.

Lemma 4.3.3 (inversion sort) *If* $\Gamma \vdash_{\mathscr{C}} s \Rightarrow s'$ *then*

- $\Gamma \vdash_{\mathscr{C}^{\Rightarrow}}$ **wf**
- $(s, s') \in \mathcal{A}_{\mathscr{C}}$

Proof *Same proof as for CTS.*

Lemma 4.3.4 (Inversion prod) *If* $\Gamma \vdash_{\mathscr{C}} (x : A) \to B \Rightarrow F$ *then*

- $\Gamma \vdash_{\mathscr{C}} A \Rightarrow s_1$
- $\Gamma, x : A \vdash_{\mathscr{C}} B \Rightarrow s_2$.
- $F \equiv_\beta s_3$
- $(s_1, s_2, s_3) \in \mathcal{R}_{\mathscr{C}}$

Proof *Same proof as for CTS.*

Lemma 4.3.5 (inversion sort check) *If* $\Gamma \vdash_{\mathscr{C}} A \Leftarrow s''$, *then there exists* s' *such that*

- $\Gamma \vdash_{\mathscr{C}} A \Rightarrow s'$
- $(s', s'') \in \mathcal{C}^*_{\mathscr{C}}$

4.3.2 From CTS to bi-directional CTS

Using well-structured derivation trees allows us to use subject reduction at level n to prove the equivalence at level $n+1$.

The lemma below is the key lemma to make the equivalence proof work. Notice our use of well-structured derivation tree hypothesis because we need subject reduction for bi-directional CTS.

Lemma 4.3.6 *Assuming* EBI_n, *for every CTS* $\mathscr{C}$ *in normal form, if* $WS_n(\Gamma \vdash_{\mathscr{C}} B : s)$ *and* $A \preceq_{\mathscr{C}} B$ *then there exists* A' *such that* $\Gamma \vdash_{\mathscr{C}} A' \Rightarrow s'$, $A \hookrightarrow^*_\beta A'$ *and* $s' \preceq_{\mathscr{C}} s$.

Proof *The transitivity rule of* $\preceq_{\mathscr{C}}$ *is an issue here. This is why we will use* $\preceq^{t^-}_{\mathscr{C}}$ *instead which defines the same subtyping relation (see Lemma 1.7.16). The fact that these two relations are equivalent is implicitly used throughout the proof.*

$\Diamond$ $\preceq^{t^-}_{\equiv_\beta}$:

	(1)	$WS_n(\Gamma \vdash_{\mathscr{C}} B : s)$	*Main hypothesis*	
	(2)	$A \preceq^{t^-}_{\mathscr{C}} B$		
	(3)	EBI_n		
	(4)	$A \equiv_\beta B$	*Inversion on* $\preceq^t_{\equiv_\beta}$	*2*
	(5)	$A \hookrightarrow^*_\beta C \hookleftarrow_\beta B$	*Confluence of* β	*4*
	(6)	$WS_n(\Gamma \vdash_{\mathscr{C}} C : s)$	*Subject reduction*	*1,5*
	(7)	$\Gamma \vdash_{\mathscr{C}} C \Leftarrow s$	EBI_n	*3*
	(8)	$\Gamma \vdash_{\mathscr{C}} C \Rightarrow s'$	*inversion sort check (4.3.5)*	*7*
✯	(9)	$s' \preceq^{t^-}_{\mathscr{C}} s$		
	(10)	*Let* $A' = C$	*Definition of* A'	
✯	(11)	$A \hookrightarrow^*_\beta A'$	*Definition of* A'	*5*
✯	(12)	$\Gamma \vdash_{\mathscr{C}} A' \Rightarrow s'$	*Definition of* A'	*8*

$\Diamond$ $\preceq^{t^-}_{\mathscr{C}^*_{\mathscr{C}}}$:

	(1)	$WS_n(\Gamma \vdash_{\mathscr{C}} B : s)$	*Main hypothesis*	
	(2)	$A \preceq^{t^-}_{\mathscr{C}} B$		
	(3)	EBI_n		
	(4)	$\mathscr{C}$ *in normal form*		
	(5)	$A \equiv_\beta s_A$	*Inversion on* $\preceq^{t^-}_{\mathscr{C}^*_{\mathscr{C}}}$	*2*
	(6)	$B \equiv_\beta s_B$		
	(7)	$s_A \preceq^{t^-}_{\mathscr{C}} s_B$		
	(8)	$B \hookrightarrow^*_\beta s_B$	*By confluence of* β	*6*
	(9)	$WS_n(\Gamma \vdash_{\mathscr{C}} s_B : s)$	*Subject reduction*	*1,8*
	(10)	$\Gamma \vdash_{\mathscr{C}} s_B \Leftarrow s$	EBI_n	*3,9*
	(11)	$\Gamma \vdash_{\mathscr{C}} s_B \Rightarrow s_D$	*inversion sort check (4.3.5)*	*10*
	(12)	$s_D \preceq^{t^-}_{\mathscr{C}} s$		
	(13)	$(s_B, s_D) \in \mathcal{A}$	*inversion sort (4.3.3)*	*11*
	(14)	$A \hookrightarrow^*_\beta s_A$	*By confluence of* β	*5*
	(15)	*Let* $A' = s_A$	*Definition of* A'	
	(16)	$\Gamma \vdash_{\mathscr{C}} s_A \Rightarrow s'$	$\mathrm{NF}_{\mathcal{A}}$	*4, 7,13*
	(17)	$s' \preceq^{t^-}_{\mathscr{C}} s_D$		
✯	(18)	$s' \preceq^{t^-}_{\mathscr{C}} s$	*Transitivity of* $\preceq^{t^-}_{\mathscr{C}}$ *(1.7.15)*	*17,12*
✯	(19)	$\Gamma \vdash_{\mathscr{C}} A' \Rightarrow s'$	*Definition of* A'	*16,15*
✯	(20)	$A \hookrightarrow^*_\beta A'$	*Definition of* A'	*14, 15*

$\Diamond$ $\preceq^{t^-}_{\Pi}$:

	(1)	$WS_n(\Gamma \vdash_{\mathscr{C}} B : s)$	*Main hypothesis*	
	(2)	$A \preceq^{t-}_{\mathscr{C}} B$		
	(3)	EBI_n		
	(4)	$\mathscr{C}$ *in normal form*		
	(5)	$A \equiv_\beta (x : A_1) \to A_2$	*Inversion on* $\preceq^{t-}_{\Pi}$	*2*
	(6)	$B \equiv_\beta (x : B_1) \to B_2$		
	(7)	$A_1 \equiv_\beta B_1$		
	(8)	$A_2 \preceq^{t-}_{\mathscr{C}} B_2$		
	(9)	$B \hookrightarrow^*_\beta (x : B_3) \to B_4 \hookleftarrow^*_\beta (x : B_1) \to B_2$	*Confluence of* β	*6*
	(10)	$A \hookrightarrow^*_\beta (x : A_3) \to A_4 \hookleftarrow^*_\beta (x : A_1) \to A_2$	*Confluence of* β	*5*
	(11)	$B_1 \equiv_\beta B_3$	*Product injectivity (1.4.2)*	*9*
	(12)	$B_2 \equiv_\beta B_4$		
	(13)	$A_1 \equiv_\beta A_3$	*Product injectivity (1.4.2)*	*10*
	(14)	$A_2 \equiv_\beta A_4$		
	(15)	$WS_n(\Gamma \vdash_{\mathscr{C}} (x : B_3) \to B_4 : s)$	*Subject reduction*	*1,9*
	(16)	$\Gamma \vdash_{\mathscr{C}} (x : B_3) \to B_4 \Leftarrow s$	EBI_n	*3, 15*
	(17)	$\Gamma \vdash_{\mathscr{C}} (x : B_3) \to B_4 \Rightarrow s_3$	*inversion sort (4.3.3)*	*16*
	(18)	$s_3 \preceq^{t-}_{\mathscr{C}} s$		
	(19)	$A_3 \hookrightarrow^*_\beta C \hookleftarrow^*_\beta B_3$	*Confluence of* β	*7,13,11*
	(20)	$WS_n(\Gamma \vdash_{\mathscr{C}} (x : C) \to B_4 : s_3)$	*Subject reduction*	*1,9,19*
	(21)	$\Gamma \vdash_{\mathscr{C}} (x : C) \to B_4 \Leftarrow s_3$	EBI_n	*1,20*
	(22)	$\Gamma, x : C \vdash_{\mathscr{C}} B_4 \Rightarrow s_2$	*Inversion prod (4.3.4)*	*21*
	(23)	$\Gamma \vdash_{\mathscr{C}} C \Rightarrow s_1$		
	(24)	$(s_1, s_2, s_3) \in \mathcal{R}_{\mathscr{C}}$		
	(25)	$A_2 \preceq^{t-}_{\mathscr{C}} B_4$	*Lemma (1.7.13)*	*8,12*
	(26)	$A_4 \preceq^{t-}_{\mathscr{C}} B_4$	*Lemma (1.7.14)*	*25,14*
	(27)	$\Gamma, x : C \vdash_{\mathscr{C}} A_{4'} \Rightarrow s_{2'}$	*Induction Hypothesis*	*22, 26*
	(28)	$s_{2'} \preceq^{t-}_{\mathscr{C}} s_2$		
	(29)	$A_4 \hookrightarrow_\beta A_{4'}$		
	(30)	*Let* $A' = (x : C) \to A_{4'}$		
	(31)	$(s_1, s_{2'}, s') \in \mathcal{R}_{\mathscr{C}}$	$\mathrm{NF}_{\mathcal{R}}$	*4,24,28*
	(32)	$(s', s_3) \in \mathcal{C}^*_{\mathscr{C}}$		
	(33)	$\Gamma \vdash_{\mathscr{C}} (x : C) \to A_{4'} \Rightarrow s'$	$\mathscr{C}^{\Rightarrow}_{\Pi}$	*23,27,31*
	(34)	$s' \preceq^{t-}_{\mathscr{C}} s_3$	$\preceq^{t-}_{\mathcal{C}^*_{\mathscr{C}}}$	*32*
★	(35)	$\Gamma \vdash_{\mathscr{C}} A' \Rightarrow s'$	*Definition of* A'	*30, 33*
★	(36)	$s' \preceq^{t-}_{\mathscr{C}} s$	*Transitivity of* $\preceq^{t-}_{\mathscr{C}}$ *(1.7.15)*	*34,18*
★	(37)	$A \hookrightarrow^*_\beta A'$	*Congruence of* β	*30,10, 19, 29*

Lemma 4.3.7 *Assuming* EBI_n *and* $\mathscr{C}$ *is in normal form:*

- *If* $WS_{n+1}(\Gamma \vdash_{\mathscr{C}} M : A)$ *then* $\Gamma \vdash_{\mathscr{C}} M \Leftarrow A$.
- *If* $WS_{n+1}(\Gamma \vdash_{\mathscr{C}} \mathbf{wf})$ *then* $\Gamma \vdash_{\mathscr{C}^{\Rightarrow}} \mathbf{wf}$

Proof *By induction on the derivation. We handle here the product case and the abstraction case which are two cases which use use Lemma 4.3.6 and therefore the fact that* C *is in normal form. The proof for the other cases are straightforward.*

◊ $\mathscr{C}_\Pi$*:* $t = (x : B) \to C, A = s$

	(1)	$WS_n(\Gamma \vdash_{\mathscr{C}} (x:B) \to C : s)$	*Main hypothesis*	
	(2)	$\Gamma \vdash_{\mathscr{C}} B : s_1$	*Inversion on* $\mathscr{C}_\Pi$	*1*
	(3)	$\Gamma, x:B \vdash_{\mathscr{C}} C : s_2$		
	(4)	$(s_1, s_2, s) \in \mathcal{R}$		
	(5)	$\Gamma \vdash_{\mathscr{C}} B \Leftarrow s_1$	*Induction Hypothesis*	*2*
	(6)	$\Gamma, x:B \vdash_{\mathscr{C}} C \Leftarrow s_2$	*Induction Hypothesis*	*3*
	(7)	$\Gamma \vdash_{\mathscr{C}} B \Rightarrow D$	*check-to-infer (4.3.2)*	*5*
	(8)	$D \preceq_{\mathscr{C}} s_1$		
	(9)	$\Gamma, x:B \vdash_{\mathscr{C}} C \Leftarrow E$	*check-to-infer (4.3.2)*	*6*
	(10)	$E \preceq_{\mathscr{C}} s_2$		
	(11)	$D \equiv_\beta s_{1'}$	*Lemma (1.4.1)*	*10*
	(12)	$s_{1'} \preceq_{\mathscr{C}} s_1$		
	(13)	$E \equiv_\beta s_{2'}$	*Lemma (1.4.1)*	*8*
	(14)	$s_{2'} \preceq_{\mathscr{C}} s_2$		
	(15)	$\Gamma \vdash_{\mathscr{C}} B \Rightarrow s_{1'}$	$\mathscr{C}^{\Rightarrow}_{\equiv_s}$	*7,11*
	(16)	$\Gamma, x:B \vdash_{\mathscr{C}} C \Rightarrow s_{2'}$	$\mathscr{C}^{\Rightarrow}_{\equiv_s}$	*9,13*
	(17)	$(s_{1'}, s_{2'}, s') \in \mathcal{R}$	$\mathrm{NF}_{\mathcal{R}}$	*4,12,14*
	(18)	$(s', s) \in \mathcal{C}$		
	(19)	$\Gamma \vdash_{\mathscr{C}} (x:B) \to C \Rightarrow s'$	$\mathscr{C}^{\Rightarrow}_{\Pi}$	*15,16,17*
	(20)	$\Gamma \vdash_{\mathscr{C}} (x:B) \to C \Leftarrow s$	$\mathscr{C}^{\Leftarrow}_{\preceq_s}$	*19,18*
★	(21)	$\Gamma \vdash_{\mathscr{C}} t \Leftarrow A$	*Definition of* t *and* A	*20*

◊ $\mathscr{C}_\lambda$: $t = \lambda x:B.\, u, A = (x:B) \to C$

	(1)	$WS_{n+1}(\Gamma \vdash_{\mathscr{C}} \lambda x:B.\, u : (x:B) \to C)$	*Main hypothesis*	
	(2)	$\mathscr{C}$ *is in normal form*		
	(3)	EBI_n		
	(4)	$WS_{n+1}(\Gamma, x:B \vdash_{\mathscr{C}} u : C)$	*Inversion on* $\mathscr{C}_\lambda$	*1*
	(5)	$WS_n(\Gamma \vdash_{\mathscr{C}} (x:B) \to C : s)$		
	(6)	$\Gamma, x:B \vdash_{\mathscr{C}} u \Leftarrow C$	*Induction Hypothesis*	*4*
	(7)	$\Gamma \vdash_{\mathscr{C}} (x:B) \to C \Leftarrow s$	*Induction Hypothesis*	*5*
	(8)	$\Gamma \vdash_{\mathscr{C}} (x:B) \to C \Rightarrow s'$	*inversion sort check (4.3.5)*	*7*
	(9)	$\Gamma, x:B \vdash_{\mathscr{C}} u \Rightarrow C'$	*check-to-infer (4.3.2)*	*6*
	(10)	$C' \preceq_{\mathscr{C}} C$		
	(11)	$WS_n(\Gamma \vdash_{\mathscr{C}} B : s_1)$	*Inversion on product*	*5*
	(12)	$WS_n(\Gamma, x:B \vdash_{\mathscr{C}} C : s_2)$		
	(13)	$(s_1, s_2, s') \in \mathcal{R}$		
	(14)	$\Gamma, x:B \vdash_{\mathscr{C}} C'' \Rightarrow s_{2'}$	*Lemma (4.3.6)*	*3,2,12,10*
	(15)	$s_{2'} \preceq_{\mathscr{C}} s_2$		
	(16)	$C' \hookrightarrow^*_\beta C''$		
	(17)	$\Gamma \vdash_{\mathscr{C}} B \Leftarrow s_1$	EBI_n	*5, 11*
	(18)	$\Gamma \vdash_{\mathscr{C}} B \Rightarrow s_{1'}$	*inversion sort check (4.3.5)*	*17*
	(19)	$s_{1'} \preceq_{\mathscr{C}} s_1$		
	(20)	$(s_{1'}, s_{2'}, s_3) \in \mathcal{R}$	$\mathrm{NF}_{\mathcal{R}}$	*2,13,19,15*
	(21)	$\Gamma \vdash_{\mathscr{C}} (x:B) \to C'' \Rightarrow s_3$	$\mathscr{C}^{\Rightarrow}_{\Pi}$	*18,14,20*
	(22)	$\Gamma, x:B \vdash_{\mathscr{C}} u \Rightarrow C''$	$\mathscr{C}^{\Leftarrow}_{\preceq}$	*9,14, 16*
	(23)	$\Gamma \vdash_{\mathscr{C}} \lambda x:B.\, u \Rightarrow (x:B) \to C''$	$\mathscr{C}^{\Rightarrow}_{\lambda}$	*22,21*
	(24)	$(x:B) \to C'' \preceq_{\mathscr{C}} (x:B) \to C$	$\preceq^r_\Pi, \preceq_{trans}$	*14,10*
	(25)	$\Gamma \vdash_{\mathscr{C}} \lambda x:B.\, u \Leftarrow (x:B) \to C$	$\mathscr{C}^{\Leftarrow}_{\preceq}$	*23,8,24*
★	(26)	$\Gamma \vdash_{\mathscr{C}} t \Leftarrow A$	*Definition of* t *and* A	*25*

Lemma 4.3.8 *For all n, we have EBI_n.*

Proof *By induction on n. The base cases are trivial. The induction step is handled by 4.3.7.*

Theorem 4.3.9 (Equivalence between typing and bi-directional typing) *For any CTS in normal form, usual typing and bi-directional typing are equivalent:*

$$\Gamma \vdash_{\mathscr{C}} t : A \Leftrightarrow \Gamma \vdash_{\mathscr{C}} t \Leftarrow A$$

$$\Gamma \vdash_{\mathscr{C}} \mathbf{wf} \Leftrightarrow \Gamma \vdash_{\mathscr{C}\Rightarrow} \mathbf{wf}$$

Proof *The right to left implication is proved in Bi-directional typing soudness (4.3.1). The left to right implication is proved in Lemma (4.3.8).*

4.4 Future Work

Larger class of CTS specification: Even if we clearly used our the two properties involved in the definition of CTS in normal form, it is not clear whether this proof could be extended for a larger class of CTS. We think however, that manipulating CTS in normal form is interesting because of Lemma 4.3.6. This is why a possible answer to this question is given by Conjecture 12: Showing that in general, given a CTS specification, there always exist an equivalent specification which is in normal form.

The well-structured hypothesis It is not clear whether the well-structured hypothesis is necessary. Another way to prove this equivalence would be to first prove subject reduction on the bi-directional CTS. However, we realized that the substitution lemma fails for the same reason: At some point we need to use subject reduction. It would be interesting to see whether applying more restrictions on the specification such as having also a functional CTS and injective CTS could work. Indeed, in [Bar99b], Gilles Barthe shows that the well-typed hypothesis of the product for the abstraction rule $\mathscr{C}_\lambda$ can be weakened in the case of PTS. Would it be the same for CTS?

Decidable type checking for CTS: As mentioned at the beginning of this chapter, the system presented here is not syntax-directed because the judgment $\Gamma \vdash_{\mathscr{C}} t \Rightarrow A$ is not a function from Γ and t. It would be interesting to know whether the CTS class we have identified (CTS in normal form) has decidable type checking. Put it in another way: Can we extend the Equivalence between typing and bi-directional typing with a syntax-directed by-directional type checking where the judgment $\Gamma \vdash_{\mathscr{C}} t \Rightarrow A$ a function of Γ and t. This is not obvious because such system breaks the symmetry of the conversion as for the expansion postponement conjecture.

Chapter 5

λΠ-calculus modulo theory as a PTS modulo

In this chapter, we introduce the λΠ-calculus modulo theory that we will use as a logical framework. The λΠ-calculus modulo theory extends the logical framework LF [HHP93b] where the notion of conversion is generalized into a congruence. This congruence is specified by a series of *equations* (or judgmental equalities) explicitly carried in the typing context of typing judgments. For interoperability, we have chosen λΠ-calculus modulo theory over LF because customizing the conversion allows us to have embeddings which are *shallow* meaning that the encodings are lighter and in practice easier to type check.

In this presentation, the custom conversion is decided by judgmental equalities which are introduced in the typing context. As we will see, this type system is very expressive but at some cost: Type checking is not always decidable in the λΠ-calculus modulo theory since it depends on the decidability of the congruence.

In the literature, the type system of the λΠ-calculus modulo theory is often refined so that instead of introducing judgmental equalities, rewrite rules are introduced instead. This trick allows to recover decidability of type checking if the rewrite rules satisfy some properties. This latter type system is implemented in the Dedukti tool that will be presented in Chapter 8.

In this work, we first describe our encodings into the λΠ-calculus modulo theory and then describe how the judgmental equalities can be turned into rewrite rules for Dedukti. Having this splitting has several advantages:

- The type system of the λΠ-calculus modulo theory as defined by Frédéric Blanqui in [Bla01] is stable. Having a type system which is stable allows to have a clear meta-theory of this type system,
- In constrast, the typing system of Dedukti evolves through time alongside its meta-theory. This means that Dedukti as a logical framework evolves. However, it is not clear today whether Dedukti will evolve into a single direction. The problem is that people are sometimes looking into generalising the notion of rewriting in Dedukti. However, it is not clear if all the features wanted by all the users are compatible with each other,
- Encodings into the λΠ-calculus modulo theory tend to be a little bit simpler and avoid tedious details related to rewrite rules,
- Once we have an encoding into the λΠ-calculus modulo theory, it is often easier to understand how it can be turned into an encoding for Dedukti.

Sorts	s	$\in \mathcal{S}$		
Terms	M, N, A, B	$\in \mathcal{T}$	$::=$	$x \mid s \mid M\ N \mid \lambda x : A.\, M \mid (x : A) \to B$
Contexts	Γ, Δ	$\in \mathcal{G}$	$::=$	$\varnothing \mid \Gamma, x : A \mid \Gamma, A \equiv_{\Delta} B$

Figure 5.1: PTS modulo syntax

In this chapter we present PTS modulo, an extension of PTS with a custom conversion. Then we present the λΠ-CALCULUS MODULO THEORY the type system generated by the **P** specification. Then we introduce *shallow embeddings* and explain why they are so important in practice. Finally, we conclude with a discussion about the meta-properties we aim to have with our encodings.

5.1 PTS modulo

5.1.1 Syntax

PTS modulo extend PTS by generelazing the conversion as congruence generated by *equations*. We do not follow the same formalization as in [Bla01] where all the equations are part of the specification. To be closer to DEDUKTI (the concrete system we use, presented in Chapter 8), we prefer to add a new construction to the typing context which allows the addition of new equations.

Definition 5.1.1 (Syntax of terms)
The syntax of terms is defined in Fig. 5.1. It extends PTS syntax with a new construction for typing contexts: $A \equiv_{\Delta} B$ meaning that for any substitution $\sigma : \Delta \to \mathcal{T}$, $A\sigma$ is convertible to $B\sigma$ if $A\sigma$ and $B\sigma$ share a common type.

5.1.2 Specification

Definition 5.1.2 (PTS modulo specification)
A PTS modulo specification *is the same as a PTS specification (1.3.1).*

5.1.3 Typing

Definition 5.1.3 (Typed substitution)
Given two typing contexts Δ and Γ, a substitution $\sigma : \Delta \to \Gamma$ is defined as a function from $Dom(\Delta) \to \mathcal{T}$ where for every $x \in Dom(\Delta)$, there exists A such that $\Gamma \vdash_{\mathscr{D}} \sigma(x) : A$.

Definition 5.1.4 (Typing of CTS)
The typing system induced by a PTS modulo specification $\mathscr{R}$ is defined in Fig. 5.3. In the rule , the notation $C[A]$ means that C is a term with a hole filled with A.

Example 5.1 *In the typing context Γ defined as:*

- $\mathbb{N} : \star$
- $Vect : \mathbb{N} \to \star$
- $+ : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$

$$\frac{A \equiv_\beta B}{A \equiv_{\beta\Gamma} B}\ \equiv_{\beta\Gamma}{}^{\beta} \qquad \frac{(M \equiv_\Delta N) \in \Gamma \qquad A = M\sigma \qquad B = N\sigma \qquad \sigma \in \Delta \to \Gamma}{A \equiv_{\beta\Gamma} B}\ \equiv_{\beta\Gamma}{}^{\Delta}$$

$$\frac{A \equiv_{\beta\Gamma} B}{B \equiv_{\beta\Gamma} A}\ \equiv_{\beta\Gamma}{}^{sym} \qquad \frac{A \equiv_{\beta\Gamma} B \qquad B \equiv_{\beta\Gamma} C}{A \equiv_{\beta\Gamma} C}\ \equiv_{\beta\Gamma}{}^{trans} \qquad \frac{A \equiv_{\beta\Gamma} B}{C[A] \equiv_{\beta\Gamma} C[B]}\ \equiv_{\beta\Gamma}{}^{[\cdot]}$$

$$\frac{A \equiv_{\beta\Gamma} B}{A\sigma \equiv_{\beta\Gamma} B\sigma}\ \equiv_{\beta\Gamma}{}^{\sigma}$$

Figure 5.2: PTS modulo congruence relation

$$\frac{}{\varnothing \vdash_{\mathscr{R}} \mathbf{wf}}\ \mathscr{R}^{\mathbf{wf}}_{\varnothing}$$

$$\frac{\Gamma \vdash_{\mathscr{R}} A : s \qquad x \notin \Gamma}{\Gamma, x : A \vdash_{\mathscr{R}} \mathbf{wf}}\ \mathscr{R}^{\mathbf{wf}}_{var}$$

$$\frac{\Gamma, \Delta \vdash_{\mathscr{R}} B : T \qquad \Gamma, \Delta \vdash_{\mathscr{R}} A : T}{\Gamma, A \equiv_\Delta B \vdash_{\mathscr{R}} \mathbf{wf}}\ \mathscr{R}^{\mathbf{wf}}_{\equiv}$$

$$\frac{\Gamma \vdash_{\mathscr{R}} \mathbf{wf} \qquad (x : A) \in \Gamma}{\Gamma \vdash_{\mathscr{R}} x : A}\ \mathscr{R}_{var}$$

$$\frac{\Gamma \vdash_{\mathscr{R}} \mathbf{wf} \qquad (s_1, s_2) \in \mathcal{A}}{\Gamma \vdash_{\mathscr{R}} s_1 : s_2}\ \mathscr{R}_{sort}$$

$$\frac{\Gamma \vdash_{\mathscr{R}} A : s_1 \qquad \Gamma, x : A \vdash_{\mathscr{R}} B : s_2 \qquad (s_1, s_2, s_3) \in \mathcal{R}}{\Gamma \vdash_{\mathscr{R}} (x : A) \to B : s_3}\ \mathscr{R}_{\Pi}$$

$$\frac{\Gamma, x : A \vdash_{\mathscr{R}} M : B \qquad \Gamma \vdash_{\mathscr{R}} (x : A) \to B : s}{\Gamma \vdash_{\mathscr{R}} \lambda x : A.\, M : (x : A) \to B}\ \mathscr{R}_{\lambda}$$

$$\frac{\Gamma \vdash_{\mathscr{R}} M : (x : A) \to B \qquad \Gamma \vdash_{\mathscr{R}} N : A}{\Gamma \vdash_{\mathscr{R}} M\ N : B\{x \leftarrow N\}}\ \mathscr{R}_{app}$$

$$\frac{\Gamma \vdash_{\mathscr{R}} M : A \qquad \Gamma \vdash_{\mathscr{R}} B : s \qquad A \equiv_{\beta\Gamma} B}{\Gamma \vdash_{\mathscr{R}} M : B}\ \mathscr{R}_{\equiv_{\beta\Gamma}}$$

Figure 5.3: Typing rules for PTS modulo

- $\times : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$
- $0 : \mathbb{N}$
- $x + x \equiv_{x:\mathbb{N}} 2 \times x$
- $y \equiv_{y:\mathbb{N}} y + 0$
- $z : \mathbb{N}$
- $t : Vect\ z + (z + 0)$

one may derive the judgment $\Gamma \vdash_{\mathscr{R}} t : Vect\ (2 \times z)$ *using the rule* $\mathscr{R}_{\equiv_{\beta\Gamma}}$. *The conversion allows the use of the judgmental equality* $y \equiv_{y:\mathbb{N}} y+0$ *with the substitution* $\{y \leftarrow z\}$ *so that we can deduce with the rule* ${\equiv_{\beta\Gamma}}^{\Delta}$ *and* ${\equiv_{\beta\Gamma}}^{\sigma}$ *that* $z \equiv_{\beta\Gamma} z+0$. *With the rule* ${\equiv_{\beta\Gamma}}^{sym}$ *we have* $z+0 \equiv_{\beta\Gamma} z$. *Using the rule* ${\equiv_{\beta\Gamma}}^{[\cdot]}$, *we can derive* $z + (z + 0) \equiv_{\beta\Gamma} z + z$. *Using the substitution* $\{x \leftarrow z\}$ *and the judgmental equality* $x + x \equiv_{x:\mathbb{N}} 2 \times x$ *we can derive with the rule* ${\equiv_{\beta\Gamma}}^{\Delta}$ *and* ${\equiv_{\beta\Gamma}}^{\sigma}$ *the judgmental equality* $z + z \equiv_{\beta\Gamma} 2 \times z$. *Finally, we can conclude with the rule* ${\equiv_{\beta\Gamma}}^{trans}$ *that* $z + (z + 0) \equiv_{\beta\Gamma} 2 \times z$.

5.1.4 Meta-theory for PTS modulo

In PTS modulo, we lose the fact the product is injective which is a property of PTS (and CTS). This property is essential to prove subject reduction. Hence, in PTS modulo, subject reduction needs to be relativized according to this property. In this chapter, we present a generalization of this property presented in Chapter 1 for CTS (Definition 1.4.2) because a term can reduce to a product. The proof of the following results can be found in [Bla01].

Definition 5.1.5 (IP)
Injectivity of product, denoted $IP(\Gamma)$, *is defined as the following property: If*

$$(x : B_1) \to C_1 \equiv_{\beta\Gamma} A \equiv_{\beta\Gamma} (x : B_2) \to C_2$$

then $B_1 \equiv_{\beta\Gamma} B_2$ *and* $C_1 \equiv_{\beta\Gamma} C_2$.

Definition 5.1.6 (SIP)
Strong injectivity of product, denoted $SIP\Gamma$, *is defined as the following property: For all* Γ' *such that* $\Gamma' \subseteq \Gamma$, *we have* $IP(\Gamma)$.

Lemma 5.1.1 (Weakening) *If* $\Gamma, \Gamma' \vdash_{\mathscr{R}} \mathbf{wf}$ *and* $\Gamma \vdash_{\mathscr{R}} t : A$ *then* $\Gamma, \Gamma' \vdash_{\mathscr{R}} t : A$.

Lemma 5.1.2 (Typing Context wf) *If* $\Gamma \vdash_{\mathscr{R}} t : A$ *then* $\Gamma \vdash_{\mathscr{R}} \mathbf{wf}$.

Lemma 5.1.3 (Well sorted) *If* $\Gamma \vdash_{\mathscr{R}} t : A$ *then* $\Gamma \vdash_{\mathscr{R}} A : s$.

Lemma 5.1.4 (Substitution lemma) *If* $\Gamma, x : A, \Gamma' \vdash_{\mathscr{R}} t : B$ *and* $\Gamma \vdash_{\mathscr{R}} N : A$ *then* $\Gamma, \Gamma' \{x \leftarrow N\} \vdash_{\mathscr{R}} t \{x \leftarrow N\} : B \{x \leftarrow N\}$.

Theorem 5.1.5 (Subject reduction for β) *For all* Γ *such that* $SIP(\Gamma)$, *if* $\Gamma \vdash_{\mathscr{R}} t : A$ *and* $t \hookrightarrow_{\beta} t'$ *then* $\Gamma \vdash_{\mathscr{R}} t' : A$.

Theorem 5.1.6 (Subject equivalence for Γ) *For all* Γ *such that* $SIP(\Gamma)$, *if* $\Gamma \vdash_{\mathscr{R}} t : A$ *and* $t \equiv_{\Gamma} t'$ *then* $\Gamma \vdash_{\mathscr{R}} t' : A$.

Remark 22 *The type system presented in Fig. 5.3 is incremental in the sense that judgmental equalities are added in the typing context one by one which requires to have the injectivity of product every time a new judgmental equality is added to the typing context. This is not needed in [Bla01] since all the equalities are a part of the specification.*

We type equalities in a similar way, except that our system is more limited for some corner cases. Our system does not allow rules which needs itself to be well-typed while in [Bla01] it is possible.

5.1.5 λΠ-CALCULUS MODULO THEORY

In this section we define one particular specification of a PTS modulo which is the λΠ-CALCULUS MODULO THEORY. The theory generated by this specification will be our framework for our theoretical embeddings. Implementations have been done with DEDUKTI (see Chapter 8).

Definition 5.1.7 (λΠ-CALCULUS MODULO THEORY)
The λΠ-CALCULUS MODULO THEORY is defined as the PTS modulo generated by the specification $\mathscr{D}$ defined below:

- $\mathcal{A} = \{(\star, \Box)\}$
- $\mathcal{R} = \{(\star, \Box, \Box), (\star, \star, \star)\}$

Remark 23 *The λΠ-CALCULUS MODULO THEORY extends λ**P** with an custom convesion.*

Our main interest for the λΠ-CALCULUS MODULO THEORY is that it can be used as a *logical framework* meaning that it can be used to express other logical theories, in our case, other type systems. Many *logical frameworks* already exist, the first one being probably predicate logic (or First-Order Logic). A seminal paper about logical frameworks is [HHP93b] which presents the PTS λ**P** (also called LF) as a logical framework. This gave rise to TWELF, a tool based upon LF. One main advtange of LF as a logical framework is that it is possible to use *Higher-Order abstract encoding* (HOAS) for embeddings without having *exotic functions*. Higher-order abstract encoding describe the fact that a binder such as $\lambda x.\, x$ can be encoded using the binder of the logical framework. In the case of the λΠ-CALCULUS MODULO THEORY it means that a binder will be encoded by a binder of the λΠ-CALCULUS MODULO THEORY. *Exotic functions* is this idea that using HOAS, the *meta* (or target) system may express more well-typed functions in the encoding than the source system. Exotic functions tend to break a property for encodings calls *conservativity* which is detailed in Section 5.3.2.

5.2 Embeddings in λΠ-CALCULUS MODULO THEORY

In the λΠ-CALCULUS MODULO THEORY, we are interested in *shallow embeddings*.

Definition 5.2.1 (Shallow encoding)
We say that an encoding from one type system $\mathcal{L}$ to another type system $\mathcal{L}'$ is shallow *if:*

- *A judgment is translated as a judgment*
- *A binder is translated as a binder or as a constant applied to some arguments and finally a binder*

There are at least two advantages to use shallow embeddings:

- when these encodings are implemented in a tool such as DEDUKTI which implements the λΠ-CALCULUS MODULO THEORY for example, encodings can be computed and type checking is scalable,

- import and export functions from and to the λΠ-CALCULUS MODULO THEORY are easier to define.

These are probably the main advantages of the λΠ-CALCULUS MODULO THEORY as a logical framework with respect to LF. Indeed, as shown in [CD07] all functional PTS have a shallow embedding into the λΠ-CALCULUS MODULO THEORY.

Theorem 5.2.1 (Cousineau & Dowek [CD07]) *For every functional PTS generated by the specification $\mathscr{P}$, there exists an embedding of $\mathscr{P}$ into $\mathscr{D}$.*

Through the remaining of this thesis, we will see several shallow embedding into the λΠ-CALCULUS MODULO THEORY. The main one being the encoding of CTS which is detailed in Chapter 6.

5.3 Meta-theory of embeddings

When we define an embedding into the λΠ-CALCULUS MODULO THEORY, we are interested in two meta-properties:

- *Soundness* which expresses that for every judgment that is derivable in the original theory its shallow encoding into the λΠ-CALCULUS MODULO THEORY is also derivable

- *Conservativity* which, roughly, expresses that the encodings cannot prove more judgments than in the original theory.

5.3.1 Soundness

Definition 5.3.1 (Soundness)
An embedding $[\![\cdot]\!]$. of a logic $\mathcal{L}$ in the λΠ-CALCULUS MODULO THEORY *is* sound *if for every judgment $\Gamma \vdash_{\mathcal{L}} t : A$ its embedding $[\![\Gamma]\!] \vdash_{\mathscr{D}} [\![t]\!] : [\![A]\!]$ is derivable.*

Soundness is a relatively easy property to prove because it is in general proved by induction on the derivation tree. The main difficulty is to ensure that any computation of the original theory is preserved through the encoding. However, we will see that in the case of CTS, the soundness proof is not easy to define because of a well-foundedness problem mentioned in Chapter 3.

Breaking the soundness property implies that some well-typed terms in the original system will not be type checkable in the λΠ-CALCULUS MODULO THEORY. This may happen in practice for derivations in the source system that use features that are not present or reflected in the target system. A concrete example is the proof irrelevance of MATITA. This feature is not translated into DEDUKTI, hence the encoding of MATITA to DEDUKTI is not sound. However, the arithmetic proofs that we translate from MATITA to DEDUKTI do not use this feature. Thus, we can show a soundness proof for a restricted version of the MATITA system into DEDUKTI. This was done by Ali Assaf in [Ass15b].

5.3.2 Conservativity

While soundness is a safe property to have for an encoding, it says little about the encoding. For example, in the typing context $\Gamma = \mathbf{I} : \star$, one can define a sound encoding such that for any Γ, t and A such that $\Gamma \vdash_{\mathcal{L}} t : A$ we have $[\![t]\!]_\Gamma = \mathbf{I}$ and $[\![A]\!]_\Gamma = \star$. This encoding is obviously sound[1]. However, this encoding is not very satisfactory because a judgment $\Gamma \vdash_{\mathcal{L}} s : A \to B$ that is not derivable in the source system is embedded as a derivable judgment. Conservativity is another property which says that any type of the logic $\mathcal{L}$ which is inhabited in the λΠ-CALCULUS MODULO THEORY is also inhabited in the logic $\mathcal{L}$.

Definition 5.3.2 (Conservativity)
An embedding $[\![\cdot]\!]_\cdot$ of a logic $\mathcal{L}$ in the λΠ-CALCULUS MODULO THEORY *is* conservative *if for every derivable judgment $[\![\Gamma]\!] \vdash_{\mathscr{D}} t : [\![A]\!]_\Gamma$, then there exists t' such that $\Gamma \vdash_{\mathcal{L}} t' : A$ is derivable in the source system.*

Conservativity discriminates trivial embeddings as the one above. For example our trivial embedding above is not conservative if $\mathcal{L}$ is a consistent logic. In general, conservativity is a much harder property to prove because it reasons on any proof term of the λΠ-CALCULUS MODULO THEORY, even the ones which are not in the image of the embedding.

However, we think that conservativity is, in the typing context of interoperability often too strong because we are often only interested in the *shape* of the type. What we mean here, is that we want to ensure that a proof of $2 + 2$ will be translated to something that looks like $2 + 2$. However, if the embedding allows to prove more theorems, it is not a real issue from an interoperability perspective. Especially in a typing context where translations are automated. Hence, to ensure that we are not defining trivial encodings as above, there is a simpler step which is to show that the shape of a type is preserved.

This means that given an embedding $[\![\cdot]\!]_\cdot$ we can define an embedding $[\![\cdot]\!]_\cdot^{-1}$ such that for all type A, we have $A \equiv_\beta [\![[\![A]\!]\,]\!]^{-1}$. Such a function is generally necessary to prove the conservativity [Ass15b]. However, it does not imply the conservativity of an encoding. Having such an inverse function allows to prove that we do not lose any information about the type, and therefore, about the mathematical statement proven.

A question we leave for futur work is whether this condition of preserving the shape is not too strong from the point of view of interoperability. Most of the time, encodings we define in DEDUKTI does not check exactly this property. Indeed, what we have in practice is a function $[\![\cdot]\!]_\cdot^{-1}$ such that $[\![[\![A]\!]\,]\!]^{-1} = A'$. And then, there exists a function f such that $f(A') \equiv_\beta A$. Such property looks like the definition of adjunction in category theory.

[1] but not shallow!

Chapter 6

Embedding CTS in λΠ-calculus modulo theory

The purpose of this chapter is to define an encoding of CTS into λΠ-calculus modulo theory. This work extends Ali Assaf's work in [Ass14] which provides an encoding of the underlying CTS specification of Matita into λΠ-calculus modulo theory. Ali Assaf's encoding has two limitations. First, it is only provided for Matita's specification. However, we would like to extend its encoding for any CTS specification so that our results could apply as well for Agda, Matita, Coq, Lean, etc... Second, his encoding eta-expands some terms, hence it breaks conservativity as it is shown in Example 6.6. Our encoding generalizes Assaf's work with an explicit *cast* operator to express subtyping between types while in Assaf's work, subtyping could be expressed only on sorts. While this generalization remains sound, we hope that it could be used to encode a more general definition of subtyping as it is implemented in Coq with Cumulative Inductive Types [TS18] or universe polymorphism [ST14].

In the same way that the type system is parametrized by a specification in CTS, our encoding is also parametrized by a λΠ-calculus modulo theory typing context. This λΠ-calculus modulo theory typing context should *implement* the CTS signature fulfilling a specification that we give in Definition 6.1.5. We will see in the second part of this thesis, that this specification can be easily implemented in Dedukti for concrete proof systems. Our encoding of CTS relies on the bi-directional type system for CTS. As such our encoding will be valid only for CTS specification in *normal form* (Definition 4.2.1). But also, we are requiring that the CTS is functional (Definition 1.3.6). These requirements are in general not too hard to satisfy since the non-functionality of a CTS can be pushed out to the cumulativity relation (see 2.2.1). Morever from this functional CTS, we can get a (weakly) equivalent CTS specification which is in normal form. In practice, these conditions are already satisfied by the specifications behind concrete systems such as the ones we have presented in Chapter 1.

Another feature of our encoding is a separation between a public and a private signature. The public signature contains the symbols used by the encoding functions. The private signature contains all the judgmental equalities necessary to prove the soundness of our encoding. The idea behind this separation is that in practice, we may have different implementations of the private signature but we want to keep one public signature to ease interoperability. We will see in the second part of this manuscript, and especially the Chapter 10 that having a common public signature makes easier interoperability between encoded proofs from Matita and Coq for example.

In the second part of this chapter, we provide a detailed soundness proof of our encoding. We present a detailed proof so that it is easy to check where and when every hypothesis are used.

We prove the soundness of our encoding only for well-structured derivation trees (see 3.1.2). Well-structured derivation trees allows the use of subject reduction for the types of terms for which we are proving the substitution lemma (see Chapter 3).

While soundness is an important property, we are aware that it is not enough in general. In particular because, as argued in Section 5.3, it does not forbid a trivial embedding which would map any type to `unit` and any term to a witness of `unit`. To demonstrate that our encoding is not trivial, we exhibit a reverse function on terms, whose composition with our embedding gives the identity function for types only. This is weaker than conservativity, but it shows that the *shape* of a type is preserved through the translation. We think that this kind of property is sufficient in a typing context of interoperability as discussed in Section 5.3.2 and conjecture that our encoding is conservative.

6.1 Description of the Embedding

Cousineau and Dowek define in [CD07] an encoding of functional PTS in λΠ-CALCULUS MODULO THEORY. However, their conservativity proof relied on the termination of the term rewrite system they used for the encoding which implies the termination of β in the original PTS (using the soundness property). Their results is extended in [Ass15a], giving a new encoding with a conservativity proof which does not rely on the termination of β in the original PTS. However, extending this encoding for CTS is not easy since CTS break an important property of PTS: Unicity of typing (Definition 1.7.12). In a CTS, a term may have several types. For example in COQ specification (see Definition 1.5.14), we have $\mathbf{0} : \mathbf{I}$ and $\mathbf{0} : \mathbf{2}$ while $\mathbf{I} \not\equiv_\beta \mathbf{2}$. Since in λΠ-CALCULUS MODULO THEORY, this property holds, it prevents our shallow embeddings to use implicit conversion for subtyping. A solution to get around this problem is to have an explicit *cast operator* for subtyping[1]. The problem with an explicit cast operator is that the encoding also needs to specify new judgmental equalities (also called *canonical* equalities). These equalities define the computational behavior of the cast operator. In Ali Assaf's work however, the *cast operator* was called *lift operator* since it could only be applied on sorts. We notice that it breaks conservativity of his encoding since he had to eta-expanse terms to handle the rule as shown in 6.6. Ali Assaf's identified three *canonical* equalities for his *lift operator*. Our work of extending his *lift operator* to a *cast operator*, requires using 8 more canonical equalities. We also follow Ali Assaf's work by using the bi-directional type system presented in Chapter 4 so that the encoding function can be expressed as a function of judgments rather than a function of derivation trees. This is closer to a concrete implementation.

An encoding in the λΠ-CALCULUS MODULO THEORY is specified by a translation function and a signature (a λΠ-CALCULUS MODULO THEORY typing context) which will be used by the encoding function. For this work, the signature of the encoding is called $\Sigma_{\mathscr{C}}$.

Definition 6.1.1 (Signature of CTS encoding to λΠ-CALCULUS MODULO THEORY) *The whole signature $\Sigma_{\mathscr{C}}$ of the embedding is split in three:*

- *The* public *signature* $\Sigma_{\mathscr{C}}^{Pu}$
- *The* specification *signature* $\Sigma_{\mathscr{C}}^{Sp}$
- *The* private *signature* $\Sigma_{\mathscr{C}}^{Pr}$

The specification signature is a parameter of our embedding, in the same way that $\mathscr{C}$ is a parameter of a CTS. However, we have to make some assumptions on this signature (see Definition 6.1.5).

[1] which extends the *lift operator* introduced by Ali Assaf in [Ass14]

The public signature is the public interface of our encoding. All the symbols used by our embedding functions are from the public signature. The private signature is the private part of our encoding. Mainly, it defines judgmental equalities between symbols of the public signature. It has to satisfy its specification given in Fig. 6.4.

We will simply write $\equiv_{\Sigma_{\mathscr{C}}}$ *for* $\equiv_{\beta\Sigma_{\mathscr{C}}}$ *(the* $\lambda\Pi$*-*CALCULUS MODULO THEORY *congruence generated by the typing context* $\Sigma_{\mathscr{C}}$*).*

Splitting the signature in three is interesting for the following reasons:

- This allows us to have some parametricity: We can switch the private part and the encoding is still working (if the private part satisfies its specification).
- We can give a soundness proof which depends only on the specification of the signature and not its concrete implementation.
- Only the public signature has to be fixed once and for all.

The last point is interesting because in practice, we often change the private signature for scalability issues or for technical details related to concrete proof systems.

The encoding function is defined on bi-directional CTS judgments. However, we define first partial encoding functions on terms and then show that these functions are total for well-typed terms (Lemma 6.1.3).

Definition 6.1.2 (CTS encoding into the $\lambda\Pi$-CALCULUS MODULO THEORY)
We define the following judgments in Fig. 6.3:

- $[t]_\Gamma$ *which translates a term* t *in a typing context* Γ *to a* DEDUKTI *term. This function is used to translate the judgment* $\Gamma \vdash_{\mathscr{C}} t \Rightarrow A$*.*
- $[t]^A_\Gamma$ *which translates a term* t *of type* A *in a typing context* Γ *to a cast term in* DEDUKTI*. The term* t *is casted from its inferred type to the type* A*. This function is used to translate the judgment* $\Gamma \vdash_{\mathscr{C}} t \Leftarrow A$*.*
- $[\![A]\!]_\Gamma$ *which translates a term* t *in a typing context* Γ *to a* DEDUKTI *type. This function corresponds to the translation of the judgment* $\Gamma \vdash_{\mathscr{C}\Rightarrow} A$ **ws***.*
- $[\![\Gamma]\!]$ *which translates a typing context* Γ *to a* DEDUKTI *typing context.*

which gives the following translation functions for judgments:

- $\Gamma \vdash_{\mathscr{C}\Rightarrow}$ **wf** *is encoded as* $\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}}$ **wf***,*
- $\Gamma \vdash_{\mathscr{C}} t \Rightarrow A$ *is encoded as* $\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [t]_\Gamma : [\![A]\!]_\Gamma$*,*
- $\Gamma \vdash_{\mathscr{C}} t \Leftarrow A$ *is encoded as* $\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [t]^A_\Gamma : [\![A]\!]_\Gamma$*.*

6.1.1 The Public Signature

We first give a description of the public signature since the other two parts rely on it as long as the encoding functions.

Definition 6.1.3 (Public signature for CTS encoding)
The public signature is defined in Fig. 6.1

$$
\begin{aligned}
\boldsymbol{\mathcal{S}} &: \star \\
\boldsymbol{s_\infty} &: \boldsymbol{\mathcal{S}} \\
\mathbf{U}_{\cdot} &: \boldsymbol{\mathcal{S}} \to \star \\
\mathbf{T}_{\cdot}\ \cdot &: (s : \boldsymbol{\mathcal{S}}) \to \mathbf{U}_s \to \star \\
\\
\mathbf{B} &: \star \\
\boldsymbol{\epsilon} &: \mathbf{B} \to \star \\
\top &: \mathbf{B} \\
\mathbf{I} &: \boldsymbol{\epsilon}\ \top \\
\forall_{\cdot}\ \cdot\ \cdot &: (s : \boldsymbol{\mathcal{S}}) \to (A : \mathbf{U}_s) \to (\mathbf{T}_s\ A \to \mathbf{B}) \to \mathbf{B} \\
\\
\boldsymbol{\mathcal{A}}(\cdot,\cdot) &: \boldsymbol{\mathcal{S}} \to \boldsymbol{\mathcal{S}} \to \mathbf{B} \\
\boldsymbol{\mathcal{R}}(\cdot,\cdot,\cdot) &: \boldsymbol{\mathcal{S}} \to \boldsymbol{\mathcal{S}} \to \boldsymbol{\mathcal{S}} \to \mathbf{B} \\
\boldsymbol{\mathcal{C}}(\cdot,\cdot) &: \boldsymbol{\mathcal{S}} \to \boldsymbol{\mathcal{S}} \to \mathbf{B} \\
\cdot\ \preceq^{\cdot}_{\cdot}\ \cdot &: (s\ s' : \boldsymbol{\mathcal{S}}) \to \mathbf{U}_s \to \mathbf{U}_{s'} \to \mathbf{B} \\
\\
\mathbf{u}_{\cdot,\cdot}\ \cdot &: (s\ s' : \boldsymbol{\mathcal{S}}) \to \boldsymbol{\epsilon}\ \boldsymbol{\mathcal{A}}(s,s') \to \mathbf{U}_{s'} \\
\boldsymbol{\pi}_{\cdot,\cdot,\cdot}\ \cdot\ \cdot\ \cdot &: (s_1\ s_2\ s_3 : \boldsymbol{\mathcal{S}}) \to \boldsymbol{\epsilon}\ \boldsymbol{\mathcal{R}}(s_1,s_2,s_3) \to \\
&\qquad (a : \mathbf{U}_{s_1}) \to (\mathbf{T}_{s_1}\ a \to \mathbf{U}_{s_2}) \to \mathbf{U}_{s_3} \\
{}^{\cdot}_{\cdot}\!\uparrow^{\cdot}_{\cdot}\ \cdot\ \cdot &: (s_1\ s_2 : \boldsymbol{\mathcal{S}}) \to (a : \mathbf{U}_{s_1}) \to (b : \mathbf{U}_{s_2}) \to \\
&\qquad \boldsymbol{\epsilon}\ \left(a\ \preceq^{s_2}_{s_1}\ b\right) \to \mathbf{T}_{s_1}\ a \to \mathbf{T}_{s_2}\ b
\end{aligned}
$$

Figure 6.1: Public signature

The first part of this signature is fairly standard. It declares a type $\boldsymbol{\mathcal{S}}$ which will be the type of the sorts. Hence, if a sort $s \in \mathcal{S}_{\mathscr{C}}$ then its *representation* in λΠ-CALCULUS MODULO THEORY should be of type $\boldsymbol{\mathcal{S}}$. Representation of sorts will be discussed in Section 6.1.3. Then we declare a special sort $\boldsymbol{s_\infty}$ which is used to give a type to top-sorts. This breaks neither consistency nor conservativity, it is just (meta) syntax to represent top-sorts[2]. We introduce a constant $\mathbf{U}$ such that $\mathbf{U}_s$ is the encoded version of CTS types living in sort s. Hence, if $\Gamma \vdash_{\mathscr{C}} A : s$ then $\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [A]_\Gamma : \mathbf{U}_s$. Finally, we introduce a constant $\mathbf{T}$ such that $\mathbf{T}_s\ A$ is the encoded version of CTS terms living in type A. Hence, if $\Gamma \vdash_{\mathscr{C}} t : A$ then $\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [A]_\Gamma : \mathbf{T}_s\ [A]_\Gamma$.

The second part of the signature is used for the cast operator. A cast from the sort s_1 to s_2 may be invalid in the original CTS. Allowing such cast in the λΠ-CALCULUS MODULO THEORY would break the conservativity. A trick would be to define the return type of the cast operator as $\max(s_1, s_2)$. However, this trick has two limitations: First, it makes the assumption that the CTS specification can be totally ordered (Definition 1.3.10). Second, other tricks are

[2]We already used this trick to define an explicit subtyping relation in Definition 3.3.1

needed such as the use of *confined* terms [ADJL16] which means any term which inhabits $\boldsymbol{\mathcal{S}}$ is convertible to a value. This hypothesis is not true when we consider extensions of CTS with universe polymorphism or cumulative inductive types. This is why we keep the return type of the cast operator as $\mathbf{U}_{s_2}$, but we require to have an irrelevant proof that $(s_1, s_2) \in \mathcal{C}^*_{\mathscr{C}}$. As we will see, the advantage for CTS is that this proof is trivial and can be achieved by computation. Moreover, we believe that such an approach can be extended when subtyping is also extended via universe polymorphism [ST14] or cumulative inductive type [TS18].

The symbol $\mathbf{B}$ is the type for meta propositions. $\boldsymbol{\epsilon}$ represents the type for irrelevant proofs for some proposition living in $\mathbf{B}$. $\top$ is a proposition inhabited by only one witness which is $\mathbf{I}$. The intention here, is that the specification signature should ensure that if $(s, s') \in \mathcal{C}^*_{\mathscr{C}}$ then $\boldsymbol{\mathcal{C}}(s, s') \equiv_{\Sigma_{\mathscr{C}}} \top$. Hence the translation function always generates $\mathbf{I}$ as a proof to check the validity of a cast operation. This check will be done by a computation.

The third part introduces symbols related to the specification of a CTS. $\boldsymbol{\mathcal{A}}(s, s')$ encodes the meta proposition $(s, s') \in \mathcal{A}_{\mathscr{C}}$, $\boldsymbol{\mathcal{R}}(s_1, s_2, s_3)$ encodes the meta proposition $(s_1, s_2, s_3) \in \mathcal{R}_{\mathscr{C}}$ and $\boldsymbol{\mathcal{C}}(s, s')$ encodes the meta proposition $(s, s') \in \mathcal{C}^*_{\mathscr{C}}$. Since subtyping is extended for products, we also have a constant $A \preceq^{s'}_{s} B$ which encodes the meta proposition that $A \preceq_{\mathscr{C}} B$.

Finally, the last part of the public signature is to encode the type constructors of a CTS. Our constructor for universes $\mathbf{u}_{s,s'}\ \mathbf{I}$ takes a proof $\mathbf{I}$ that $(s, s') \in \mathcal{A}_{\mathscr{C}}$. This is not mandatory since we are encoding functional CTS. However, it makes the computational behavior of our encoding simpler and this is heavily used by our tool Universo presented in Chapter 10. In the same way, we have a constructor for products $\boldsymbol{\pi}_{s_1,s_2,s_3}\ \mathbf{I}\ A\ (\lambda x : \mathbf{T}_{s_1}\ A.\, B)$ which encodes a product $(x : A) \to B$ with $(s_1, s_2, s_3) \in \mathcal{R}_{\mathscr{C}}$. The last constructor is the explicit cast operator ${}^{s'}_{s}\!\uparrow^{B}_{A}\ \mathbf{I}\ t$ which encodes the subtyping rule: t is of type A and is seen of type B where $A \preceq_{\mathscr{C}} B$. Again, we use the witness $\mathbf{I}$ to ensure that $A \preceq_{\mathscr{C}} B$.

6.1.2 Encoding functions

As mentioned before, we need to encode the subtyping rule in Dedukti explicitly. However, since subtyping is implicit in CTS, to ensure the soundness of the translation, the latter cannot be done directly on the judgment $\Gamma \vdash_{\mathscr{C}} t : A$ since there is no subtyping information. To solve this issue, we follow Ali Assaf's steps and translate bi-directional CTS. In bi-directional CTS, the inference judgment $\Gamma \vdash_{\mathscr{C}} t \Rightarrow A$ is encoded as usual and the checking judgment $\Gamma \vdash_{\mathscr{C}} t \Leftarrow A$ is encoded with a cast operator since by definition, the last rule of such judgment is a subtyping rule. However, this assumes that the CTS specifications need to be in normal form (Definition 4.2.1).

Another way to solve these issues would be to define directly the translation functions on derivation trees. However, we experienced that the proofs get really complicated and needed to ensure that two derivations of the same judgment give rise to two λΠ-calculus modulo theory terms which are convertible.

Before defining the translation functions, we define a predicate $\Gamma \vdash_{\mathscr{C}} A \overset{?}{\Rightarrow} s$ which is similar to the predicate $\Gamma \vdash_{\mathscr{C}_{\Rightarrow}} A$ **ws** but include the sort s_∞. This way every type is *well-sorted*, even top sorts. It is used to handle in the same way the translation of type A of sort s and a top-sort $s' \in \mathcal{S}^{\top}_{\mathscr{C}}$.

Definition 6.1.4 (Well-sorted predicate)
We define the predicate $\Gamma \vdash_{\mathscr{C}} A \overset{?}{\Rightarrow} s$ *in Fig 6.2.*

Our main reason to introduce this new notation is the following lemma:

Lemma 6.1.1 *If* $\Gamma \vdash_{\mathscr{C}_{\Rightarrow}}$ **wf** *then for all* $s \in \mathcal{S}$ *there exists* s' *such that* $\Gamma \vdash_{\mathscr{C}} s \overset{?}{\Rightarrow} s'$.

$$\frac{s \in \mathcal{S}^{\top}_{\mathscr{C}}}{\Gamma \vdash_{\mathscr{C}} s \overset{?}{\Rightarrow} \boldsymbol{s_\infty}} \overset{?}{\Rightarrow}^{top} \qquad \frac{(s, s') \in \mathcal{A}_{\mathscr{C}}}{\Gamma \vdash_{\mathscr{C}} s \overset{?}{\Rightarrow} s'} \overset{?}{\Rightarrow}^{ax} \qquad \frac{\Gamma \vdash_{\mathscr{C}} A \Rightarrow s}{\Gamma \vdash_{\mathscr{C}} A \overset{?}{\Rightarrow} s} \overset{?}{\Rightarrow}^{type}$$

Figure 6.2: Variant rules for well-sorted types in bi-directional CTS

Proof *By case analysis on* $s \in \mathcal{S}^{\top}_{\mathscr{C}}$.

Lemma 6.1.2 *If* $\Gamma \vdash_{\mathscr{C}} A$ **ws** *then there exists* $s \in \mathcal{S} \cup s_\infty$ *such that* $\Gamma \vdash_{\mathscr{C}} A \overset{?}{\Rightarrow} s$.

Proof *By case analysis on* $\Gamma \vdash_{\mathscr{C}} A$ **ws**.

The definition of our encoding functions over terms is partial because of the side-conditions. However, it is complete for well-typed terms.

Lemma 6.1.3 (Well-defined embedding)

- *If* $\Gamma \vdash_{\mathscr{C}} t \Rightarrow A$ *then* $[t]_\Gamma$ *is well-defined*
- *If* $\Gamma \vdash_{\mathscr{C}} t \Leftarrow A$ *then* $[t]^A_\Gamma$ *is well-defined*
- *If* $\Gamma \vdash_{\mathscr{C}} A \overset{?}{\Rightarrow} s$ *then* $[\![A]\!]_\Gamma$ *is well-defined*
- *If* $\Gamma \vdash_{\mathscr{C}\Rightarrow}$ **wf** *then* $[\![\Gamma]\!]$ *is well-defined*

Proof *By inversion on the derivation.*

Remark 24 *In the soundness proof we will not explicitly mention the use of a translation function is well-defined. Also, we may refer to the typing judgment which means that implicitly we have used Well-defined embedding (6.1.3) and the fact that encoded terms were well-defined.*

Example 6.1 *In the* Simply Typed Lambda Calculus, *one may derive the following judgment:* $A : \star \vdash_{\rightarrow} \lambda x : A.\, x \Rightarrow A \to A$. *Its translation to the* λΠ-calculus modulo theory *gives* $\Sigma^{Sp}_{\rightarrow}, A : \mathbf{T}_\Box\ (\mathbf{u}_{\star,\Box}\ \mathbf{I}) \vdash_{\mathscr{D}} \lambda x : \mathbf{T}_\star\ A.\, x : \mathbf{T}_\star\ (\boldsymbol{\pi}_{\star,\star,\star}\ \mathbf{I}\ A\ (\lambda x : \mathbf{T}_\star\ A.\, A))$.

Using only the public signature, this signature is ill-typed:

- *We need* $\boldsymbol{\mathcal{R}}(\star,\star,\star) \equiv_{\Sigma_{\mathscr{C}}} \top$ *and* $\boldsymbol{\mathcal{A}}(\star,\Box) \equiv_{\Sigma_{\mathscr{C}}} \top$. *This is the purpose of the specification signature.*
- *The type of the abstraction is not a product in the* λΠ-calculus modulo theory. *We will see that the private signature makes the term* $\mathbf{T}_\star\ (\boldsymbol{\pi}_{\star,\star,\star}\ \mathbf{I}\ A\ (\lambda x : \mathbf{T}_\star\ A.\, A))$ *convertible with* $(x : \mathbf{T}_\star\ A) \to \mathbf{T}_\star\ A$.

With some optimization, the judgment could be shorter: $\Sigma^{Sp}_{\rightarrow}, A : \mathbf{U}_\star \vdash_{\mathscr{D}} \lambda x : \mathbf{T}_\star\ A.\, x : \mathbf{T}_\star\ A \to \mathbf{T}_\star\ A$. *We will not consider such optimization here even if they are used in practice.*

$$
\begin{aligned}
[x]_\Gamma &= x \\
[s]_\Gamma &= \mathbf{u}_{s,s'}\ \mathbf{I} \\
&\text{when } \Gamma \vdash_{\mathscr{C}} s \overset{?}{\Rightarrow} s' \\
[M\ N]_\Gamma &= [M]_\Gamma\ [N]^A_\Gamma \\
&\text{when } \Gamma \vdash_{\mathscr{C}} M \Rightarrow (x:A) \to B \\
[\lambda x:A.\, M]_\Gamma &= \lambda x : [\![A]\!]_\Gamma .\, [M]_{\Gamma,x:A} \\
[(x:A) \to B]_\Gamma &= \boldsymbol{\pi}_{s_1,s_2,s_3}\ \mathbf{I}\ [A]_\Gamma\ (\lambda x : [\![A]\!]_\Gamma .\, [B]_{\Gamma,x:A}) \\
&\text{when } \Gamma \vdash_{\mathscr{C}} A \Rightarrow s_1 \\
&\qquad \Gamma \vdash_{\mathscr{C}} B \Rightarrow s_2 \\
&\qquad (s_1,s_2,s_3) \in \mathcal{R}
\end{aligned}
$$

$$
\begin{aligned}
[M]^B_\Gamma &= {}^{s2\uparrow^{[B]_\Gamma}_{[A]_\Gamma}}_{s1}\ \mathbf{I}\ [M]_\Gamma \\
&\text{when } \Gamma \vdash_{\mathscr{C}} M \Rightarrow A \\
&\qquad \Gamma \vdash_{\mathscr{C}} A \overset{?}{\Rightarrow} s_1 \\
&\qquad \Gamma \vdash_{\mathscr{C}} B \overset{?}{\Rightarrow} s_2
\end{aligned}
$$

$$
\begin{aligned}
[\![A]\!]_\Gamma &= \mathbf{T}_s\ [A]_\Gamma \\
&\text{when } \Gamma \vdash_{\mathscr{C}} A \overset{?}{\Rightarrow} s
\end{aligned}
$$

$$
\begin{aligned}
[\![\varnothing]\!] &= \varnothing \\
[\![\Gamma, x:A]\!] &= [\![\Gamma]\!], x : [\![A]\!]_\Gamma
\end{aligned}
$$

Figure 6.3: CTS translation functions

Example 6.2 *In the specification $\mathscr{C}^L$ (the* Calculus of Constructions *with an infinite and cumulative hierarchy of universes, Definition 1.5.13), we may derive the judgment $\vdash_{\mathscr{C}^L} \lambda x:\mathbf{0}.\, x \Leftarrow \mathbf{0} \to \mathbf{1}$. The translation of this judgment depends on the type A we infer for $\lambda x:\mathbf{0}.\, x$. If we assume that we have a proof $\vdash_{\mathscr{C}^c} \lambda x:\mathbf{0}.\, x \Rightarrow \mathbf{0} \to \mathbf{0}$ then we can translate this judgment as*

$$
\Sigma^{Sp}_{\mathscr{C}^L} \vdash_{\mathscr{D}} \mathbf{2}_{\mathbf{1}}{}^{\uparrow\begin{pmatrix}\boldsymbol{\pi}_{\mathbf{1},\mathbf{2},\mathbf{2}}\ \mathbf{I}\ \left(\mathbf{u}_{\mathbf{0},\mathbf{1}}\ \mathbf{I}\right)\ \left(\lambda x:\mathbf{T}_{\mathbf{1}}\ \left(\mathbf{u}_{\mathbf{0},\mathbf{1}}\ \mathbf{I}\right).\, \mathbf{u}_{\mathbf{1},\mathbf{2}}\ \mathbf{I}\right)\right)\\ \left(\boldsymbol{\pi}_{\mathbf{1},\mathbf{1},\mathbf{1}}\ \mathbf{I}\ \left(\mathbf{u}_{\mathbf{0},\mathbf{1}}\ \mathbf{I}\right)\ \left(\lambda x:\mathbf{T}_{\mathbf{1}}\ \left(\mathbf{u}_{\mathbf{0},\mathbf{1}}\ \mathbf{I}\right).\, \mathbf{u}_{\mathbf{0},\mathbf{1}}\ \mathbf{I}\right)\right)\end{pmatrix}}\ \mathbf{I}\ \left(\lambda x:\mathbf{T}_{\mathbf{1}}\ \left(\mathbf{u}_{\mathbf{0},\mathbf{1}}\ \mathbf{I}\right).\, x\right) : \mathbf{T}_{\star}\ \left(\boldsymbol{\pi}_{\star,\star,\star}\ \mathbf{I}\ \left(\mathbf{u}_{\mathbf{0},\mathbf{1}}\ \mathbf{I}\right)\ \left(\lambda x:\mathbf{T}_{\mathbf{1}}\ \left(\mathbf{u}_{\mathbf{0},\mathbf{1}}\ \mathbf{I}\right).\, \mathbf{u}_{\mathbf{1},\mathbf{2}}\ \mathbf{I}\right)\right)
$$

.

This translation may be scary at first, but again using some optimizations we could generate this shorter judgment instead: $\Sigma^{Sp}_{\mathscr{C}^L} \vdash_{\mathscr{D}} \lambda x:\mathbf{U}_{\mathbf{0}}.\, \mathbf{2}_{\mathbf{1}}{}^{\uparrow\begin{pmatrix}\mathbf{u}_{\mathbf{1},\mathbf{2}}\ \mathbf{I}\\ \mathbf{u}_{\mathbf{0},\mathbf{1}}\ \mathbf{I}\end{pmatrix}}\ \mathbf{I}\ x : \mathbf{U}_{\mathbf{0}} \to \mathbf{U}_{\mathbf{1}}$. To get this judgment, we have used a canonical equality which allows to permute a cast operator and an abstraction (see Section 6.1.4).

In bi-directional type systems, one cannot use subtyping directly on a variable as it is done in the judgment above but this is possible in CTS. Hence, these two judgments correspond to two different derivation trees for the judgment $\vdash_{\mathscr{C}^L} \lambda x:\mathfrak{O}.\, x : \mathfrak{O} \to \mathfrak{I}$*, one where subtyping is used on* x *and one where it is used on* $\lambda x:\mathfrak{O}.\, x$*. The idea behind canonical equalities is therefore to make these two derivation trees equal.*

6.1.3 The specification signature

The specification signature is a parameter of our encoding. For one particular CTS specification, we have one particular λΠ-CALCULUS MODULO THEORY signature $\Sigma^{Sp}_{\mathscr{C}}$. The set $\mathcal{S}$ is part of the specification, hence this specification signature needs to specify how a sort should be represented in λΠ-CALCULUS MODULO THEORY. Actually, every sort should be translated to the λΠ-CALCULUS MODULO THEORY as their representation. We will not explicitly manipulate this representation because this tends to obscure the notations and just assume that a sort s is translated as s. In practice, it is not an issue to find such a representation because the set $\mathcal{S}$ is countable. The definition below states the specification we assume in our soundness proof.

Definition 6.1.5 (Valid specification signature)
Given a specification $\mathscr{C}$*, we say that the signature* $\Sigma^{Sp}_{\mathscr{C}}$ *is* valid *denoted* $\Sigma^{Pu}_{\mathscr{C}} \models \Sigma^{Sp}_{\mathscr{C}}$ *if and only if:*

$$\Sigma^{Pu}_{\mathscr{C}}, \Sigma^{Sp}_{\mathscr{C}} \vdash_{\mathscr{D}} \mathbf{wf} \qquad (\mathbf{wf}_{\Sigma^{Sp}_{\mathscr{C}}})$$

$$\boldsymbol{\mathcal{A}}(s,s') \equiv_{\Sigma^{Sp}_{\mathscr{C}}} \top \iff (s,s') \in \mathcal{A}_{\mathscr{C}} \qquad (\boldsymbol{\mathcal{A}}_{\Sigma^{Sp}_{\mathscr{C}}})$$

$$\boldsymbol{\mathcal{A}}(s,\boldsymbol{s_\infty}) \equiv_{\Sigma^{Sp}_{\mathscr{C}}} \top \iff s \in \mathcal{S}^{\top}_{\mathscr{C}} \qquad (\boldsymbol{\mathcal{A}}_{\boldsymbol{s_\infty}\Sigma^{Sp}_{\mathscr{C}}})$$

$$\boldsymbol{\mathcal{R}}(s,s',s'') \equiv_{\Sigma^{Sp}_{\mathscr{C}}} \top \iff (s,s',s'') \in \mathcal{R}_{\mathscr{C}} \qquad (\boldsymbol{\mathcal{R}}_{\Sigma^{Sp}_{\mathscr{C}}})$$

$$\boldsymbol{\mathcal{C}}(s,s') \equiv_{\Sigma^{Sp}_{\mathscr{C}}} \top \iff (s,s') \in \mathcal{C}^{*}_{\mathscr{C}} \qquad (\boldsymbol{\mathcal{C}}_{\Sigma^{Sp}_{\mathscr{C}}})$$

Moreover we assume that $\Sigma^{Sp}_{\mathscr{C}}$ *does not break the injectivity of product in* λΠ-CALCULUS MODULO THEORY*(see Definition 5.1.5). This is more a technical restriction than a real constraint.*

In practice such specification can be satisfied easily (see Chapter 5 for concrete examples).

6.1.4 The Private Signature

The so-called *private signature* aims to contain all the judgmental equalities that should hold so that we can prove the soundness theorems. In DEDUKTI, such equalities are implemented by rewrite rules. The advantage of having a presentation with equalities instead of rewrite rules is that it is easier to give a specification for the private signature (no need to orient the equalities). The related private signature in DEDUKTI is presented in Section 8.3.

Definition 6.1.6 (Valid private signature)
A private signature is valid *denoted* $\Sigma^{Pu}_{\mathscr{C}} \models \Sigma^{Pr}_{\mathscr{C}}$ *if and only if:*

- *It does not break the injectivity of product (see Definition 5.1.5)*
- *It satifies all the equations presented in Fig. 6.4.*

Remark 25 *In this framework, since we have* $\boldsymbol{s_\infty}$*, the symbol* $\mathbf{U}_s$ *could be defined as* $\mathbf{U}_s := \mathbf{T}_{s'}\ \mathbf{u}_{s,s'}\ \mathbf{I}$ *when* $\Gamma \vdash_{\mathscr{C}} s \overset{?}{\Rightarrow} s'$*. However, it is still required to give a type to* $\mathbf{T}$*.*

$$\mathbf{T}.\ (\mathbf{u}_{s,\cdot}\ \cdot) \equiv_{\Sigma_{\mathscr{C}}} \mathbf{U}_s \qquad (\mathbf{T}-s)$$

$$\mathbf{T}.\ (\boldsymbol{\pi}_{s_1,s_2,\cdot}\ \cdot\ a\ b) \equiv_{\Sigma_{\mathscr{C}}} (x : \mathbf{T}_{s_1}\ a) \to \mathbf{T}_{s_2}\ (b\ x) \qquad (\mathbf{T}-\pi)$$

$$\mathbf{T}.\ (\vdots\uparrow^{\cdot}_{s}\ \cdot\ t) \equiv_{\Sigma_{\mathscr{C}}} \mathbf{T}_s\ t \qquad (\mathbf{T}-\uparrow)$$

$$X\ \preceq^{\cdot}_{\cdot}\ X \equiv_{\Sigma_{\mathscr{C}}} \top \qquad (\mathrm{st}-\equiv)$$

$$\mathbf{u}_{s,\cdot}\ \cdot\ \preceq^{\cdot}_{\cdot}\ \mathbf{u}_{s',\cdot}\ \cdot \equiv_{\Sigma_{\mathscr{C}}} \boldsymbol{\mathcal{C}}(s,s') \qquad (\mathrm{st}-s)$$

$$(\boldsymbol{\pi}_{s_1,s_2,\cdot}\ \cdot\ A\ B)\ \preceq^{\cdot}_{\cdot}\ (\boldsymbol{\pi}_{\cdot,s_{2'},\cdot}\ \cdot\ A\ B') \equiv_{\Sigma_{\mathscr{C}}} \forall_{s_1}\ A\ \left(\lambda x : \cdot.\ B\ x\ \preceq^{s_{2'}}_{s_2}\ B'\ x\right) \qquad (\mathrm{st}-\pi)$$

$$\left(\vdots\uparrow^{\cdot}_{\mathbf{u}_{s,\cdot}}\ \cdot\ \cdot\ A\right)\ \preceq^{s'}_{\cdot}\ B \equiv_{\Sigma_{\mathscr{C}}} A\ \preceq^{s'}_{s}\ B \qquad (\mathrm{st}-\uparrow-l)$$

$$A\ \preceq^{\cdot}_{s}\ \left(\vdots\uparrow^{\cdot}_{\mathbf{u}_{s',\cdot}}\ \cdot\ \cdot\ B\right) \equiv_{\Sigma_{\mathscr{C}}} A\ \preceq^{s'}_{s}\ B \qquad (\mathrm{st}-\uparrow-r)$$

$$\vdots\uparrow^{a}_{a}\ \cdot\ t \equiv_{\Sigma_{\mathscr{C}}} t \qquad (\uparrow-\mathrm{id})$$

$$\vdots\uparrow^{c}_{b}\ \cdot\ \left(\vdots\uparrow^{b}_{a}\ \cdot\ t\right) \equiv_{\Sigma_{\mathscr{C}}} \vdots\uparrow^{c}_{a}\ \cdot\ t \qquad (\uparrow-\uparrow)$$

$$\vdots\uparrow^{s'_3}_{s_3}\ \cdot\ (\boldsymbol{\pi}_{s_1,s_2,s_3}\ \cdot\ a\ b) \equiv_{\Sigma_{\mathscr{C}}} \boldsymbol{\pi}_{s'_1,s'_2,s'_3}\ \cdot\ \left(\vdots\uparrow^{s'_1}_{\cdot}\ \cdot\ a\right)\ \left(\lambda x.\ \vdots\uparrow^{s'_2}_{\cdot}\ \cdot\ (b\ x)\right) \qquad (\pi-\uparrow)$$

$$\vdots\uparrow^{\left(\boldsymbol{\pi}_{\cdot,s_3,\cdot}\ \cdot\ A\ C\right)}_{\left(\boldsymbol{\pi}_{s_1,s_2,\cdot}\ \cdot\ A\ B\right)}\ \cdot\ (\lambda x.\ b\ x) \equiv_{\Sigma_{\mathscr{C}}} \lambda x : \mathbf{T}_{s_1}\ A.\ \left({}^{s_3}_{s_2}\uparrow^{(C\ x)}_{(B\ x)}\ \cdot\ b\ x\right) \qquad (\uparrow-\mathrm{lam})$$

$${}^{s_3}_{s_2}\uparrow^{(C\ a)}_{(B\ a)}\ \cdot\ (b\ a) \equiv_{\Sigma_{\mathscr{C}}} \left(\vdots\uparrow^{\left(\boldsymbol{\pi}_{\cdot,s_3,\cdot}\ \cdot\ A\ C\right)}_{\left(\boldsymbol{\pi}_{\cdot,s_2,\cdot}\ \cdot\ A\ B\right)}\ \cdot\ b\right)\ a \qquad (\uparrow-\mathrm{app})$$

$$\vdots\uparrow^{\left(\vdots\uparrow^{\cdot}_{(\mathbf{u}_{s_2,\cdot}\ \cdot)}\ \cdot\ B\right)}_{A}\ \cdot\ a \equiv_{\Sigma_{\mathscr{C}}} {}^{s_2}_{s_1}\uparrow^{B}_{A}\ \cdot\ a \qquad (\uparrow_{\uparrow})$$

$$\vdots^{s_2}\uparrow^{B}_{\left(\vdots\uparrow^{\cdot}_{(\mathbf{u}_{s_1,\cdot}\ \cdot)}\ \cdot\ A\right)}\ \cdot\ a \equiv_{\Sigma_{\mathscr{C}}} {}^{s_2}_{s_1}\uparrow^{B}_{A}\ \cdot\ a \qquad (\uparrow^{\uparrow})$$

Figure 6.4: Private signature specification

The first three rules are called *decoding* rules. They give an interpretation for every type constructor. One can check with the following examples that each of these equalities is needed (except $\uparrow - \mathrm{id}$ as discussed below). The CTS specification we use here is that of LEAN (Definition 1.5.13). It is just a cumulative hierarchy of universes where $\mathbb{0}$ is predicative.

Example 6.3 *For each of the following CTS judgments, their encoding is well-typed λΠ-CALCULUS MODULO THEORY by using the judgmental equality specified on their right.*

$$\begin{array}{ll}
\vdash_{\mathscr{C}} \mathbb{0} \Rightarrow \mathbb{1} & \mathbf{T} - s \\
\vdash_{\mathscr{C}} \lambda x : \mathbb{0}.\, \mathbb{0} \Rightarrow (x : \mathbb{0}) \to \mathbb{1} & \mathbf{T} - \pi \\
f : (y : \mathbb{1} \to \mathbb{0}) \to y\ \mathbb{0} \vdash_{\mathscr{C}} f\ (\lambda x : \mathbb{1}.\, x) \Rightarrow (\lambda x : \mathbb{1}.\, x)\ \mathbb{0} & \mathbf{T} - \uparrow
\end{array}$$

The next five rules in Fig 6.4 of the private signature check that a type A is a subtype of another type B. One can check that each of these rules are needed on the following examples:

Example 6.4 *One can do the same exercise as in the previous example(s) for the following judgments:*

$$\begin{array}{ll}
\vdash_{\mathscr{C}} \mathbb{0} \Leftarrow \mathbb{1} & \mathrm{st} - \equiv \\
\vdash_{\mathscr{C}} \mathbb{0} \Leftarrow \mathbb{2} & \mathrm{st} - s \\
\vdash_{\mathscr{C}} \lambda x : \mathbb{0}.\, \mathbb{0} \Leftarrow \mathbb{0} \to \mathbb{2} & \mathrm{st} - \pi \\
f : (y : \mathbb{1} \to \mathbb{0}) \to y\ \mathbb{0} \vdash_{\mathscr{C}} f\ (\lambda x : \mathbb{1}.\, x) \Leftarrow (\lambda x : \mathbb{1}.\, x)\ \mathbb{2} & \mathrm{st} - \uparrow - l\ ,\ \mathrm{st} - \uparrow - r
\end{array}$$

Finally, the last seven rules are called *canonicity* rules and allow to permute a cast operator with the other type constructors. Except the identity cast ($\uparrow - \mathrm{id}$), one can check that all these rules are needed on the following examples.

Example 6.5 *Using the following typing context,*

$$\begin{aligned}
\Gamma = {} & p : \mathbb{2} \to \mathbb{2}, \\
& f : (c : \mathbb{1}) \to p\ c \to \mathbb{0}, \\
& g : (a : \mathbb{2}) \to p\ (a \to a)
\end{aligned}$$

one can do the same exercise as in the previous example(s) for the following judgments.

$$\begin{array}{ll}
\Gamma \vdash_{\mathscr{C}} f\ (\mathbb{0} \to \mathbb{0})\ (g\ \mathbb{0}\ \mathbb{0}) \Rightarrow \mathbb{0} & \pi - \uparrow \\
\Gamma \vdash_{\mathscr{C}} f\ (\mathbb{0} \to \mathbb{0})\ (g\ ((\lambda x : \mathbb{1}.\, x)\ \mathbb{0})\ \mathbb{0}) \Rightarrow \mathbb{0} & \pi - \uparrow \quad \uparrow - \uparrow \\
\vdash_{\mathscr{C}} \mathbb{0} \Rightarrow (\lambda x : \mathbb{3} \to \mathbb{3}.\, x)\ (\lambda z : \mathbb{2}.\, z)\ \mathbb{1} & \uparrow - \mathrm{app} \\
\vdash_{\mathscr{C}} \mathbb{0} \Rightarrow (\lambda x : \mathbb{3} \to \mathbb{3}.\, x\ \mathbb{1})\ (\lambda z : \mathbb{2}.\, z) & \uparrow - \mathrm{lam} \\
\vdash_{\mathscr{C}} \mathbb{0} \Rightarrow (\lambda x : \mathbb{1} \to \mathbb{3}.\, x)\ (\lambda z : ((\lambda y : \mathbb{3}.\, y)\ \mathbb{1}).\, \mathbb{2})\ \mathbb{1} & \uparrow^{\uparrow} \\
\vdash_{\mathscr{C}} \mathbb{0} \Rightarrow (\lambda x : ((\lambda z : \mathbb{4}.\, z)\ \mathbb{1} \to \mathbb{2}).\, x)\ (\lambda y : \mathbb{1}.\, \mathbb{2})\ \mathbb{1} & \uparrow_{\uparrow}
\end{array}$$

The canonicity equality $\uparrow - \mathrm{id}$ is interesting, and we detail in Section 6.4 why it is not present in the examples above.

Defining encodings function where a cast is added to every applications makes the translated terms huge and too long to be type checked in practice. In particular, most of the time an identity cast is introduced while it is unnecessary. In the following example $\mathbb{N} : \mathbb{0}, 0 : \mathbb{N} \vdash_{\mathscr{C}} (\lambda x : \mathbb{N}.\, x)\ 0 \Rightarrow \mathbb{N}$, we know that $\mathbb{N}$ is always a subtype of itself, so there is no need to add an identity cast around 0. This leads to an optimization which is to remove identity casts during the translation.

Then, we conjecture that this optimization requires using identity casts. We have also observed that identity casts under this optimization are used extensively. Adding identity casts are also helpful for interoperability, especially for our tool Universo presented in Chapter 10.

6.2 Soundness

This section aims at giving a soundness prove of our encoding. In particular, we aim at proving the following results:

- If $\Gamma \vdash_{\mathscr{C}} t \Rightarrow A$ then $\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [t]_{\Gamma} : [\![A]\!]_{\Gamma}$
- If $\Gamma \vdash_{\mathscr{C}} t \Leftarrow A$ then $\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [t]^{A}_{\Gamma} : [\![A]\!]_{\Gamma}$
- If $\Gamma \vdash_{\mathscr{C}} \mathbf{wf}$ then $\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} \mathbf{wf}$

As mentioned previously, we need however to use well-structured derivation trees in order to prove that the encoding functions permute with substitutions.

This soundness proof will assume that the initial judgment has a well-structured derivation tree (Definition 3.1.2). This hypothesis is needed because to prove that the encoding functions permutes with substitutions, we need the preservation of computation the type of the term being translated. Well-structured derivation trees precisely allow us to do that (see Chapter 3). We think that we could also have used an explicitly typed subtyping relation as discussed in Section 3.3.

However, we have not defined well-structured derivation trees if it is derived in the bi-directional typing system.

Definition 6.2.1 (Well-structured derivation trees for bi-directional CTS)
Let us denote ϕ the computable function defined by the Embedding Theorem 4.3.8. Then we say that a derivation tree π in bi-directional CTS is well-structured if it is the image by the function ϕ of a well-structured derivation tree. In other words:

$$WS(\pi) := WS(\pi') \wedge \phi(\pi') = \pi$$

We also define being well-structured at level n for bi-directional derivation trees as follow:

$$WS_n(\pi) := WS_n(\pi') \wedge \phi(\pi') = \pi$$

These notations are extended naturally for judgments as we did for CTS. One may check that, because of the equivalence theorem (Theorem 4.3.8), this definition is compatible with the properties defined in Definition 3.1.2 for well-structured derivation trees.

Finally, for all the reasons we have explained above, the theorem we prove in this section is:

Theorem 6.2.1 *Given a function specification $\mathscr{C}$ in normal form, a specification signature $\Sigma^{Sp}_{\mathscr{C}}$ and a private signature $\Sigma^{Pr}_{\mathscr{C}}$ if we have*

- *No clash between the symbol names of the encoding and variable names*
- *A valid specification signature: $\Sigma^{Pu}_{\mathscr{C}} \models \Sigma^{Sp}_{\mathscr{C}}$ (see Definition 6.1.5)*
- *A valid private signature: $\Sigma^{Pu}_{\mathscr{C}} \models \Sigma^{Pr}_{\mathscr{C}}$ (see Definition 6.1.6)*

then we have for all n,

- *If* $WS_n(\Gamma \vdash_{\mathscr{C}} t \Rightarrow A)$ *then* $\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [t]_\Gamma : [\![A]\!]_\Gamma$
- *If* $WS_n(\Gamma \vdash_{\mathscr{C}} t \Leftarrow A)$ *then* $\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [t]^A_\Gamma : [\![A]\!]_\Gamma$
- *If* $WS_n(\Gamma \vdash_{\mathscr{C}\Rightarrow}$ **wf**$)$ *then* $\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}}$ **wf**

Throughout the proof we will not mention these hypothesis for every statement. Instead, we will refer to them using the following assumptions:

Assumption 1 *From now on, we make the following assumptions:*

$$\forall x \in \mathcal{V},\ x \notin \Sigma_{\mathscr{C}} \qquad (\mathcal{V} \cap \Sigma_{\mathscr{C}} = \emptyset)$$
$$\Sigma^{Pu}_{\mathscr{C}} \models \Sigma^{Sp}_{\mathscr{C}} \qquad \text{(specif. sig. valid)}$$
$$\Sigma^{Pu}_{\mathscr{C}} \models \Sigma^{Pr}_{\mathscr{C}} \qquad \text{(private. sig. valid)}$$

Lemma 6.2.2 *If* $\Sigma_{\mathscr{C}}, \Gamma \vdash_{\mathscr{D}}$ **wf** *then we have injectivity of product:* $IP(\Gamma)$ *(5.1.5).*

Proof *A direct consequence of Assumption 1.*

Warning

To maintain a readable proof, the trade-off we have chosen in this manuscript is to have a detailed proof for the two key lemmas which are:

- preservation of computation 6.2.24
- preservation of typability 6.2.39

and for the other *helper* lemmas, we only sketch a proof.
In particular, we were careful in Lemma 6.2.39 to check that when the conversion rule $\mathscr{R}_{\equiv_{\beta\Gamma}}$ is used with an equation $A \equiv_{\Sigma_{\mathscr{C}}} B$ to ensure that B is well-sorted. This detail is sometimes omitted and we realized this may lead to erroneous proofs.

6.2.1 Extended meta-theory for bi-directional CTS

The soundness proof we present below uses some classic result of CTS for bi-directional CTS. We reference here these results, whose proofs are a direct consequence of the meta-theory of CTS presented in Chapter 1 and the equivalence theorem 4.3.9 between CTS and bi-directional CTS presented in Chapter 4.

We extend the notion of a type being well-sorted for bi-directional CTS.

Definition 6.2.2 (Well-sorted)
We introduce the judgment $\Gamma \vdash_{\mathscr{C}\Rightarrow} A$ **ws** *in Figure 6.5 expressing that* A *is well-sorted: Either* A *is a sort or it has a type which is a sort.*

Lemma 6.2.3 (Well-sorted ⇒) *If* $\Gamma \vdash_{\mathscr{C}} t \Rightarrow A$ *then* $\Gamma \vdash_{\mathscr{C}\Rightarrow} A$ **ws**.

Lemma 6.2.4 (Well-sorted ⇐) *If* $\Gamma \vdash_{\mathscr{C}} t \Leftarrow A$ *then* $\Gamma \vdash_{\mathscr{C}\Rightarrow} A$ **ws**.

$$\frac{}{\Gamma \vdash_{\mathscr{C}^{\Rightarrow}} s \ \mathbf{ws}}\ \mathbf{ws}^{\Rightarrow}\text{-SORT} \qquad \frac{\Gamma \vdash_{\mathscr{C}} A \Rightarrow s}{\Gamma \vdash_{\mathscr{C}^{\Rightarrow}} A \ \mathbf{ws}}\ \mathbf{ws}^{\Rightarrow}\text{-TYPE}$$

Figure 6.5: Derivation rules of well-sorted types for bi-directional CTS

Lemma 6.2.5 (Well-formed wf) *If* $\Gamma \vdash_{\mathscr{C}} t \Rightarrow A$ *then* $\Gamma \vdash_{\mathscr{C}^{\Rightarrow}} \mathbf{wf}$.

Lemma 6.2.6 (Substitution lemma)

- *If* $\Gamma, x : A, \Gamma' \vdash_{\mathscr{C}} t \Rightarrow B$ *and* $\Gamma \vdash_{\mathscr{C}} N \Leftarrow A$, *then* $\Gamma, \Gamma'\{x \leftarrow N\} \vdash_{\mathscr{C}} t\{x \leftarrow N\} \Leftarrow B\{x \leftarrow N\}$.
- *If* $\Gamma, x : A, \Gamma' \vdash_{\mathscr{C}^{\Rightarrow}} \mathbf{wf}$ *and* $\Gamma \vdash_{\mathscr{C}} N \Leftarrow A$ *then* $\Gamma, \Gamma'\{N \leftarrow A\} \vdash_{\mathscr{C}^{\Rightarrow}} \mathbf{wf}$

Proof *Corollary of Lemma 1.7.8 and Lemma 4.3.9.*

6.2.2 A lighter notation for casts

The purpose of this section is to defined a lighter notation for the cast operator (see Notation 22).

Lemma 6.2.7 (Signature well-formed) *The signature is well-formed:* $\Sigma_{\mathscr{C}} \vdash_{\mathscr{D}} \mathbf{wf}$.

Proof *By induction on the typing context. Everything is straightforward except the rule* $\uparrow -\mathrm{app}$ *because the arity of the cast operator is not the same on the left side as on the right side.*

Lemma 6.2.8 *If* $x : A \in \Gamma$ *then* $x : [\![A]\!]_{\Gamma} \in [\![\Gamma]\!]$.

Proof *By induction on* Γ.

Lemma 6.2.9 *If* $\mathcal{A}(s, s') \equiv_{\Sigma_{\mathscr{C}}} \top$ *then* $\Sigma_{\mathscr{C}} \vdash_{\mathscr{D}} \mathbf{u}_{s,s'}\ \mathbf{I} : \mathbf{U}_{s'}$

Lemma 6.2.10 *If* $\Sigma_{\mathscr{C}}^{Pu} \models \Sigma_{\mathscr{C}}^{Sp}$, *then for all* $s \in \mathcal{S}$, *there exists* s' *such that* $\Sigma_{\mathscr{C}} \vdash_{\mathscr{D}} \mathbf{u}_{s,s'}\ \mathbf{I} : \mathbf{U}_{s'}$

Proof *By case analysis on* $s \in \mathcal{S}_{\mathscr{C}}^{\top}$.

◊ $s \in \mathcal{S}_{\mathscr{C}}^{\top}$:

	(1)	$s \in \mathcal{S}_{\mathscr{C}}^{\top}$	*Main Hypothesis*	
	(2)	$\Sigma_{\mathscr{C}}^{Pu} \models \Sigma_{\mathscr{C}}^{Sp}$		
	(3)	$\mathcal{A}(s, s_{\infty}) \equiv_{\Sigma_{\mathscr{C}}} \top$	$\mathcal{A}_{s_{\infty}\,\Sigma_{\mathscr{C}}^{Sp}}$ *(6.1.5)*	*1,2*
✯	(4)	$\Sigma_{\mathscr{C}} \vdash_{\mathscr{D}} \mathbf{u}_{s,s_{\infty}}\ \mathbf{I} : \mathbf{U}_{s_{\infty}}$	*Lemma (6.2.9)*	*3*

◊ $s \notin \mathcal{S}_{\mathscr{C}}^{\top}$:

	(1)	$s \notin \mathcal{S}_{\mathscr{C}}^{\top}$	*Main Hypothesis*	
	(2)	$\Sigma_{\mathscr{C}}^{Pu} \models \Sigma_{\mathscr{C}}^{Sp}$		
	(3)	$\exists s', (s, s') \in \mathcal{A}$	*Not a top-sort*	*1,2*
	(4)	$\mathcal{A}(s, s') \equiv_{\Sigma_{\mathscr{C}}} \top$	$\mathcal{A}_{\Sigma_{\mathscr{C}}^{Sp}}$ *(6.1.5)*	*3*
✯	(5)	$\Sigma_{\mathscr{C}} \vdash_{\mathscr{D}} \mathbf{u}_{s,s'}\ \mathbf{I} : \mathbf{U}_{s'}$	*Lemma (6.2.9)*	*4*

Notation 22 *Assuming that* $\Gamma \vdash_{\mathscr{D}} A : \mathbf{U}_{s_A}$ *and* $\Gamma \vdash_{\mathscr{D}} B : \mathbf{U}_{s_B}$ *we define the following notation:*

$$\uparrow_A^B \mathbf{I}\ t := {}^{s_B}_{s_A}\uparrow_A^B \mathbf{I}\ t$$

when A *and* B *are both sorts, we also write* $\uparrow_{s_1}^{s_2} \mathbf{I}\ t$ *instead of* $\uparrow_{\mathbf{u}_{s_1,s_1'}}^{\mathbf{u}_{s_2,s_2'}} \mathbf{I}\ \mathbf{I}\ t$*. This is well-defined thanks to Lemma (6.2.10) and because* $\mathscr{C}$ *is functional.*

6.2.3 Preservation of computation

Preservation of computation is the most tedious part. We show that if $A \equiv_\beta B$ then their encoding is also convertible, namely $[\![A]\!]_\Gamma \equiv_{\Sigma_\mathscr{C}} [\![B]\!]_\Gamma$. In particular, the key lemma will be to show that the encoding functions permute with substitutions (Lemma 6.2.25). These lemmas are tedious to prove but straightforward.

Lemma 6.2.11 *If* $\Gamma \vdash_\mathscr{C} t \Rightarrow A$ *and* $\Gamma, \Gamma' \vdash_\mathscr{C} \mathbf{wf}$ *then* $[t]_\Gamma = [t]_{\Gamma,\Gamma'}$

Proof *By induction on* Γ'*.*

Lemma 6.2.12 *If* $\Gamma \vdash_\mathscr{C} t \Leftarrow A$ *and* $\Gamma, \Gamma' \vdash_\mathscr{C} \mathbf{wf}$ *then* $[t]^A_\Gamma = [t]^A_{\Gamma,\Gamma'}$

Proof *By induction on* Γ'*.*

Lemma 6.2.13 *If* $\Gamma \vdash_\mathscr{C} A \overset{?}{\Rightarrow} s$ *and* $(s, s') \in \mathcal{C}^*_\mathscr{C}$ *then* $\mathbf{T}_s\ [A]_\Gamma \equiv_{\Sigma_\mathscr{C}} \mathbf{T}_{s'}\ [A]^{s'}_\Gamma$*.*

Proof *A consequence of the canonicity rule* $\mathbf{T}-\uparrow$ *(6.1.6).*

Lemma 6.2.14 *If* $\Gamma \vdash_\mathscr{C} (x:A) \to B \Rightarrow s$ *then* $[\![(x:A) \to B]\!]_\Gamma \equiv_{\Sigma_\mathscr{C}} (x : [\![A]\!]_\Gamma) \to [\![B]\!]_{\Gamma,x:A}$*.*

Proof *A consequence of the canonicity rule* $\mathbf{T} - \pi$ *(6.1.6).*

Lemma 6.2.15 (id-cast) *If* $\Sigma^{Pu}_\mathscr{C} \models \Sigma^{Pr}_\mathscr{C}$, $\Gamma \vdash_\mathscr{C} t \Rightarrow A$ *then* $[t]_\Gamma \equiv_{\Sigma_\mathscr{C}} [t]^A_\Gamma$*.*

Proof *A consequence of the canonicity rule* $\uparrow - \mathrm{id}$ *(6.1.6).*

Lemma 6.2.16 (π-cast) *If* $\Sigma^{Pu}_\mathscr{C} \models \Sigma^{Pr}_\mathscr{C}$, $\Gamma \vdash_\mathscr{C} (x:A) \to B \Leftarrow s$, $\Gamma \vdash_\mathscr{C} A \Rightarrow s_1$ *and* $\Gamma, x : A \vdash_\mathscr{C} B \Rightarrow s_2$ *then* $[(x:A) \to B]^s_\Gamma \equiv_{\Sigma_\mathscr{C}} (x : [A]^{s_1}_\Gamma) \to [B]^{s_2}_{\Gamma,x:A}$*.*

Proof *A consequence of the canonicity rule* $\pi - \uparrow$ *(6.1.6).*

Lemma 6.2.17 (app-cast) *If* $\Sigma^{Pu}_\mathscr{C} \models \Sigma^{Pr}_\mathscr{C}$, $\Gamma \vdash_\mathscr{C} f\ a \Leftarrow D\{x \leftarrow a\}$, $\Gamma \vdash_\mathscr{C} f \Rightarrow (x:A) \to B$ *and* $\Gamma \vdash_\mathscr{C} a \Rightarrow A$ *then* $[f\ a]^D_\Gamma \equiv_{\Sigma_\mathscr{C}} [f]^{(x:A)\to D}_\Gamma\ [a]^A_\Gamma$*.*

Proof *A consequence of the canonicity rules* $\uparrow - \mathrm{app}$ *,* $\uparrow_\uparrow$ *and* $\uparrow^\uparrow$ *(6.1.6).*

Lemma 6.2.18 (λ-cast) *If* $[A]_\Gamma\ [\sigma] \equiv_{\Sigma_\mathscr{C}} [A\sigma]^s_\Gamma$ *then* $[\![A]\!]_\Gamma\ [\sigma] \equiv_{\Sigma_\mathscr{C}} [\![A\sigma]\!]_\Gamma$*.*

Proof *By definition of* $[\![A]\!]_\Gamma$*.*

Lemma 6.2.19 *If* $\Gamma \vdash_{\mathscr{C}\Rightarrow} \mathbf{wf}$ *then* $[\![s]\!]_\Gamma \equiv_{\Sigma_\mathscr{C}} \mathbf{U}_s$*.*

The definitions below are related to well-structured derivation trees. They are used to derive an induction principle compatible with subject reduction as we did in the previous chapters.

Definition 6.2.3 (SUBST)
We denote $SUBST_n$ *the following property:*

- *If* $WS_n(\Gamma, x : A, \Gamma' \vdash_{\mathscr{C}} M \Rightarrow B)$ *and* $\Gamma \vdash_{\mathscr{C}} N \Leftarrow A$, *then*

$$[M]_{\Gamma,x:A,\Gamma'}\left\{x \leftarrow [N]^{A}_{\Gamma}\right\} \equiv_{\Sigma_{\mathscr{C}}} [M\{x \leftarrow N\}]^{B\{x\leftarrow N\}}_{\Gamma,\Gamma'\{x\leftarrow N\}}$$

- *If* $WS_n(\Gamma, x : A, \Gamma' \vdash_{\mathscr{C}} M \Leftarrow B)$ *and* $\Gamma \vdash_{\mathscr{C}} N \Leftarrow A$, *then*

$$[M]^{B}_{\Gamma,x:A,\Gamma'}\left\{x \leftarrow [N]^{A}_{\Gamma}\right\} \equiv_{\Sigma_{\mathscr{C}}} [M\{x \leftarrow N\}]^{B\{x\leftarrow N\}}_{\Gamma,\Gamma'\{x\leftarrow N\}}$$

Because during the proof of Lemma 6.2.22 we need to use the preservation of computation, we also use the following definition.

Definition 6.2.4 (CONV)
We denote $CONV_n$ *as the following property: If* $WS_{n+1}(\Gamma \vdash_{\mathscr{C}} t \overset{?}{\Rightarrow} A)$, $\Gamma \vdash_{\mathscr{C}} B \overset{?}{\Rightarrow} s$, $A \equiv_\beta B$ *then* $[t]_\Gamma \equiv_{\Sigma_{\mathscr{C}}} [t]^{B}_{\Gamma}$.

Lemma 6.2.20 *Assuming* $SUBST_n$, *if* $WS_n(\Gamma \vdash_{\mathscr{C}} t \Rightarrow A)$ *and* $t \hookrightarrow_\beta t'$ *then* $[t]_\Gamma \equiv_{\Sigma_{\mathscr{C}}} [t']^{A}_{\Gamma}$.

Proof *By induction on* $t \hookrightarrow_\beta t'$. *The base case is handled by the* $SUBST_n$ *hypothesis.*

Lemma 6.2.21 *Assuming* $SUBST_n$, *if* $WS_n(\Gamma \vdash_{\mathscr{C}} t \Rightarrow A)$ *and* $t \hookrightarrow^*_\beta t'$ *then* $[t]_\Gamma \equiv_{\Sigma_{\mathscr{C}}} [t']^{A}_{\Gamma}$.

Proof *By induction on the length of the reduction of* $t \hookrightarrow^*_\beta t'$ *and the rule* $\uparrow - \uparrow$.

Lemma 6.2.22 *We have* $SUBST_n$ *implies* $CONV_n$.

Proof *This proof is mainly a direct consequence of Lemma 6.2.21. However, we would like to draw the attention on a crucial point to make this proof work. Assuming the last derivation rule is* $\mathscr{C}^{\Leftarrow}_{\geq}$ *and that* $\Gamma \vdash_{\mathscr{C}} t : A$, *we want to prove from* $[t]_\Gamma$, $[B]_\Gamma s$ *and* $A \equiv_\beta B$ *the fact that we have* $[t]_\Gamma \equiv_{\Sigma_{\mathscr{C}}} [t]^{B}_{\Gamma}$. *The idea is therefore to add an identity cast* $\uparrow - \mathrm{id}$.

The identity cast that we would like to add is $\uparrow^{[B]_\Gamma}_{[A]_\Gamma} \mathbf{I}\ [t]_\Gamma$. *But, there is a catch because we cannot justify that* $[A]_\Gamma \equiv_{\Sigma_{\mathscr{C}}} [B]_\Gamma$! *Indeed, what we can show using Lemma 6.2.21 is that there exists* s *and* s' *such that* $[A]_\Gamma \equiv_{\Sigma_{\mathscr{C}}} [C]^{s}_{\Gamma}$ *and* $[B]_\Gamma \equiv_{\Sigma_{\mathscr{C}}} [C]^{s'}_{\Gamma}$. *But a priori* $[C]^{s}_{\Gamma}$ *and* $[C]^{s'}_{\Gamma}$ *are different.*

Our solution is first to add the identity cast $\uparrow^{[C]^{s}_{\Gamma}}_{[A]_\Gamma} \mathbf{I}\ [t]_\Gamma$, *then using the rule* $\uparrow^{\uparrow}$ *we obtain* $\uparrow^{[C]_\Gamma}_{[A]_\Gamma} \mathbf{I}\ [t]_\Gamma$, *then by transitivity using the rule* $\uparrow^{\uparrow}$ *again we obtain* $\uparrow^{[C]^{s'}_{\Gamma}}_{[A]_\Gamma} \mathbf{I}\ [t]_\Gamma$ *and finally we get* $\uparrow^{[B]_\Gamma}_{[A]_\Gamma} \mathbf{I}\ [t]_\Gamma$.

So indeed, we can justify that $[t]_\Gamma$ *is convertible to* $\uparrow^{[B]_\Gamma}_{[A]_\Gamma} \mathbf{I}\ [t]_\Gamma$ *without having* $[A]_\Gamma$ *convertible to* $[B]_\Gamma$! *In* Dedukti, *because we use a confluent rewrite system,* $[A]_\Gamma$ *and* $[B]_\Gamma$ *will actually be convertible but this is not guarantded by the private signature.*

Lemma 6.2.23 *We have* $SUBST_0$

Proof *The cases* $\mathscr{C}^{\Rightarrow}_{\equiv}$ *and* $\mathscr{C}^{\Rightarrow}_{\equiv^s}$ *are not possible. The other cases are the same as in Lemma 6.2.24.*

Lemma 6.2.24 *We have* $SUBST_n$ *implies* $SUBST_{n+1}$.

Proof *By induction on* $\Gamma, x : A, \Gamma' \vdash_{\mathscr{C}} M \Rightarrow B$. *For clarity we will use the following notations:*

- $\sigma = \{x \leftarrow N\}$
- $[\sigma] = \left\{x \leftarrow [N]_{\Gamma}^{A}\right\}$

◊ $\mathscr{C}^{\Rightarrow}_{var}$: $M = y$

By case analysis on $x = y$

□ $x = y$: $B = A$

(1)	$\Gamma, x : A, \Gamma' \vdash_{\mathscr{C}} x \Rightarrow A$	*Main hypothesis*	
(2)	$\Gamma \vdash_{\mathscr{C}} N \Leftarrow A$		
(3)	$[x]_{\Gamma,x:A,\Gamma'} = x$	*Definition of* $[\cdot]$.	
(4)	$[x]_{\Gamma,x:A,\Gamma'}[\sigma] = [N]_{\Gamma}^{A}$	*Substitution*	3
(5)	$[x]_{\Gamma,x:A,\Gamma'}[\sigma] = [x\sigma]_{\Gamma}^{A}$	*Substitution*	4
(6)	$[x]_{\Gamma,x:A,\Gamma'}[\sigma] = [x\sigma]_{\Gamma}^{A\sigma}$	*Substitution* ($x \notin \mathrm{FV}(A)$)	5
(7)	$\Gamma, x : A, \Gamma' \vdash_{\mathscr{C}\Rightarrow}$ wf	*Inversion on* $\mathscr{C}^{\Rightarrow}_{var}$	1
(8)	$\Gamma, \Gamma'\sigma \vdash_{\mathscr{C}\Rightarrow}$ wf	*Substitution lemma (6.2.6)*	7,2
(9)	$[x\sigma]_{\Gamma}^{A\sigma} = [x\sigma]_{\Gamma,\Gamma'\sigma}^{A\sigma}$	*Lemma (6.2.12)*	2,8
(10)	$[x]_{\Gamma,x:A,\Gamma'}[\sigma] = [x\sigma]_{\Gamma,\Gamma'\sigma}^{A\sigma}$	*Transitivity of* $=$	6,9
(11)	$[x]_{\Gamma,x:A,\Gamma'}[\sigma] \equiv_{\Sigma_{\mathscr{C}}} [x\sigma]_{\Gamma,\Gamma'\sigma}^{A\sigma}$	*Reflexivity of* $\equiv$	10
★ (12)	$[M]_{\Gamma,x:A,\Gamma'}[\sigma] \equiv_{\Sigma_{\mathscr{C}}} [M\sigma]_{\Gamma,\Gamma'\sigma}^{B\sigma}$	*Definition* M, y, B	11

□ $x \neq y$:

(1)	$\Gamma, x : A, \Gamma' \vdash_{\mathscr{C}} y \Rightarrow B$	*Main hypothesis*	
(2)	$\Gamma \vdash_{\mathscr{C}} N \Leftarrow A$		
(3)	$[y]_{\Gamma,x:A,\Gamma'} = y$	*Definition of* $[\cdot]$.	
(4)	$[y]_{\Gamma,x:A,\Gamma'}[\sigma] = y$	*Substitution*	3
(5)	$[y]_{\Gamma,x:A,\Gamma'}[\sigma] \equiv_{\Sigma_{\mathscr{C}}} y$	*Reflexivity of* $\equiv_{\Sigma_{\mathscr{C}}}$	4
(6)	$\Gamma, x : A, \Gamma' \vdash_{\mathscr{C}\Rightarrow}$ wf	*Inversion on* $\mathscr{C}^{\Rightarrow}_{var}$	1
(7)	$y \in (\Gamma, x : A, \Gamma')$		
(8)	$y \in (\Gamma, \Gamma')$	$x \neq y$	7
(9)	$\Gamma, \Gamma'\sigma \vdash_{\mathscr{C}\Rightarrow}$ wf	*Substitution lemma (6.2.6)*	6,2
(10)	$y : B\sigma \in (\Gamma, \Gamma'\sigma)$	*Typing Context Substitution*	8
(11)	$\Gamma, \Gamma'\sigma \vdash_{\mathscr{C}} y \Rightarrow B\sigma$	$\mathscr{C}^{\Rightarrow}_{var}$	9,10
(12)	$y = [y]_{\Gamma,\Gamma'\sigma}$	*Definition of* $[\cdot]$.	
(13)	$y \equiv_{\Sigma_{\mathscr{C}}} [y]_{\Gamma,\Gamma'\sigma}$	*Reflexivity of* $\equiv_{\Sigma_{\mathscr{C}}}$	12
(14)	$\Sigma_{\mathscr{C}}^{Pu} \models \Sigma_{\mathscr{C}}^{Pr}$	*Assumption 1*	
(15)	$[y]_{\Gamma,\Gamma'\sigma} \equiv_{\Sigma_{\mathscr{C}}} [y]_{\Gamma,\Gamma'\sigma}^{B\sigma}$	*id-cast (6.2.15)*	14,11
(16)	$[y]_{\Gamma,x:A,\Gamma'}[\sigma] \equiv_{\Sigma_{\mathscr{C}}} [y]_{\Gamma,\Gamma'\sigma}^{B\sigma}$	*Transitivity of* $\equiv_{\Sigma_{\mathscr{C}}}$	5,13,15
(17)	$[y]_{\Gamma,x:A,\Gamma'}[\sigma] \equiv_{\Sigma_{\mathscr{C}}} [y\sigma]_{\Gamma,\Gamma'\sigma}^{B\sigma}$	*Substitution*	16
★ (18)	$[M]_{\Gamma,x:A,\Gamma'}[\sigma] \equiv_{\Sigma_{\mathscr{C}}} [M\sigma]_{\Gamma,\Gamma'\sigma}^{B\sigma}$	*Definition of* M	17

◊ $\mathscr{C}^{\Rightarrow}_{sort}$: $M = s, B = s'$

	(1)	$\Gamma, x : A, \Gamma' \vdash_{\mathscr{C}} s \Rightarrow s'$	*Main hypothesis*	
	(2)	$\Gamma \vdash_{\mathscr{C}} N \Leftarrow A$		
	(3)	$[s]_{\Gamma,x:A,\Gamma'} = \mathbf{u}_{s,s'}\ \mathbf{I}$	*Definition of* $[\cdot]$.	1
	(4)	$[s]_{\Gamma,x:A,\Gamma'}[\sigma] = \mathbf{u}_{s,s'}\ \mathbf{I}$	*Substitution*	3
	(5)	$\Gamma, x : A, \Gamma' \vdash_{\mathscr{C}}$ wf	*Inversion on* $\mathscr{C}^{\Rightarrow}_{sort}$	1
	(6)	$(s, s') \in \mathcal{A}_{\mathscr{C}}$		
	(7)	$\Gamma, \Gamma'\sigma \vdash_{\mathscr{C}^{\Rightarrow}}$ wf	*Substitution lemma (6.2.6)*	5,2
	(8)	$\Gamma, \Gamma'\sigma \vdash_{\mathscr{C}} s \Rightarrow s'$	$\mathscr{C}^{\Rightarrow}_{sort}$	7,6
	(9)	$\mathbf{u}_{s,s'}\ \mathbf{I} = [s]_{\Gamma,\Gamma'\sigma}$	*Definition of* $[\cdot]$.	8
	(10)	$\Sigma^{Pu}_{\mathscr{C}} \models \Sigma^{Pr}_{\mathscr{C}}$	*Assumption 1*	
	(11)	$[s]_{\Gamma,\Gamma'\sigma} = [s]^{s'}_{\Gamma,\Gamma'\sigma}$	*id-cast (6.2.15)*	10,8
	(12)	$[s]_{\Gamma,x:A,\Gamma'}[\sigma] = [s]^{s'}_{\Gamma,\Gamma'\sigma}$	*Transitivity of* $=$	4,9,11
	(13)	$[s]_{\Gamma,x:A,\Gamma'}[\sigma] = [s\sigma]^{s'\sigma}_{\Gamma,\Gamma'\sigma}$	*Substitution*	12
	(14)	$[s]_{\Gamma,x:A,\Gamma'}[\sigma] \equiv_{\Sigma_{\mathscr{C}}} [s\sigma]^{s'\sigma}_{\Gamma,\Gamma'\sigma}$	*Reflexivity of* $\equiv_{\Sigma_{\mathscr{C}}}$	12
★	(15)	$[M]_{\Gamma,x:A,\Gamma'}[\sigma] \equiv_{\Sigma_{\mathscr{C}}} [M\sigma]^{B\sigma}_{\Gamma,\Gamma'\sigma}$	*Definition of* M, B	13

◊ $\mathscr{C}^{\Rightarrow}_{\Pi}$: $M = (x : C) \to D, B = s$

	(1)	$WS_{n+1}(\Gamma, x : A, \Gamma' \vdash_{\mathscr{C}} (y : C) \to D \Rightarrow s)$	*Main hypothesis*	
	(2)	$\Gamma \vdash_{\mathscr{C}} N \Leftarrow A$		
	(3)	$WS_{n+1}(\Gamma, x : A, \Gamma' \vdash_{\mathscr{C}} C \Rightarrow s_1)$	*Inversion on* $\mathscr{C}^{\Rightarrow}_{\Pi}$	1
	(4)	$WS_{n+1}(\Gamma, x : A, \Gamma', y : C \vdash_{\mathscr{C}} D \Rightarrow s_2)$		
	(5)	$(s_1, s_2, s) \in \mathcal{R}_{\mathscr{C}}$		
	(6)	$([C]_{\Gamma,x:A,\Gamma'})[\sigma] \equiv_{\Sigma_{\mathscr{C}}} [C\sigma]^{s_1\sigma}_{\Gamma,\Gamma'\sigma}$	*Induction Hypothesis*	3,2
	(7)	$([D]_{\Gamma,x:A,\Gamma',y:C})[\sigma] \equiv_{\Sigma_{\mathscr{C}}} [D\sigma]^{s_2\sigma}_{\Gamma,\Gamma'\sigma,y:C\sigma}$	*Induction Hypothesis*	4,2
	(8)	$[(y : C) \to D]_{\Gamma,x:A,\Gamma'} = (y : [C]_{\Gamma,x:A,\Gamma'}) \to [D]_{\Gamma,x:A,\Gamma',y:C}$	*Definition of* $[\cdot]$.	3,4,5
	(9)	$([(y : C) \to D]_{\Gamma,x:A,\Gamma'})[\sigma] = (y : ([C]_{\Gamma,x:A,\Gamma'})[\sigma]) \to ([D]_{\Gamma,x:A,\Gamma',y:C})[\sigma]$	*Substitution*	8
	(10)	$([(y : C) \to D]_{\Gamma,x:A,\Gamma'})[\sigma] \equiv_{\Sigma_{\mathscr{C}}} (y : [C\sigma]^{s_1\sigma}_{\Gamma,\Gamma'\sigma}) \to [D\sigma]^{s_2\sigma}_{\Gamma,\Gamma'\sigma,y:C\sigma}$	*Congruence of* $\equiv_{\Sigma_{\mathscr{C}}}$	9,6,7
	(11)	$(y : [C\sigma]^{s_1}_{\Gamma,\Gamma'\sigma}) \to [D\sigma]^{s_2}_{\Gamma,\Gamma'\sigma,y:C\sigma} \equiv_{\Sigma_{\mathscr{C}}} [((y : C) \to D)\,\sigma]^{s}_{\Gamma,\Gamma'\sigma}$	*π-cast (6.2.16)*	
	(12)	$([(y : C) \to D]_{\Gamma,x:A,\Gamma'})[\sigma] \equiv_{\Sigma_{\mathscr{C}}} [((y : C) \to D)\,\sigma]^{s}_{\Gamma,\Gamma'\sigma}$	*Congruence of* $\equiv_{\Sigma_{\mathscr{C}}}$	10,11
	(13)	$([(y : C) \to D]_{\Gamma,x:A,\Gamma'})[\sigma] \equiv_{\Sigma_{\mathscr{C}}} [((y : C) \to D)\,\sigma]^{s\sigma}_{\Gamma,\Gamma'\sigma}$	*Substitution*	12
★	(14)	$[M]_{\Gamma,x:A,\Gamma'}[\sigma] \equiv_{\Sigma_{\mathscr{C}}} [M\sigma]^{B\sigma}_{\Gamma,\Gamma'\sigma}$	*Definition of* $M, B, \sigma, [\sigma]$	13

◊ $\mathscr{C}^{\Rightarrow}_{\lambda}$: $M = \lambda y : C.\, u, B = (y : C) \to D, \sigma := \{x \leftarrow N\}.[\sigma] := \{x \leftarrow [N]^{A}_{\Gamma}\}$

	(1)	$WS_{n+1}(\Gamma, x:A, \Gamma' \vdash_{\mathscr{C}} \lambda y:C.\, u \Rightarrow (y:C) \to D)$	*Main hypothesis*	
	(2)	$\Gamma \vdash_{\mathscr{C}} N \Leftarrow A$		
	(3)	$WS_{n+1}(\Gamma, x:A, \Gamma', y:C \vdash_{\mathscr{C}} u \Rightarrow D)$	*Inversion on* $\mathscr{C}^{\Rightarrow}_{\lambda}$	1
	(4)	$WS_n(\Gamma, x:A, \Gamma' \vdash_{\mathscr{C}} (y:C) \to D \Rightarrow s)$		
	(5)	$([u]_{\Gamma,x:A,\Gamma',y:C})\,[\sigma] \equiv_{\Sigma_{\mathscr{C}}} [u\sigma]^{D\sigma}_{\Gamma,\Gamma'\sigma,y:C\sigma}$	*Induction Hypothesis*	3,2
	(6)	$([(y:C) \to D]_{\Gamma,x:A,\Gamma'})\,[\sigma] \equiv_{\Sigma_{\mathscr{C}}} [((y:C) \to D)\,\sigma]^{s\sigma}_{\Gamma,\Gamma'\sigma}$	*Induction Hypothesis*	4,2
	(7)	$[\lambda y:C.\, u]_{\Gamma,x:A,\Gamma'} = \lambda y:[\![C]\!]_{\Gamma,x:A,\Gamma'}.\,[u]_{\Gamma,x:A,\Gamma',y:C}$	*Definition of* $[\cdot]$.	1
	(8)	$([\lambda y:C.\, u]_{\Gamma,x:A,\Gamma'})\,[\sigma] = \lambda y:[\![C]\!]_{\Gamma,x:A,\Gamma'}\,[\sigma]\,.\,[u]_{\Gamma,x:A,\Gamma',y:C}\,[\sigma]$	*Substituion*	7
	(9)	$[(y:C) \to D)\,\sigma]^{s\sigma}_{\Gamma,\Gamma'\sigma} \equiv_{\Sigma_{\mathscr{C}}} (y:[C\sigma]^{s_1}_{\Gamma,\Gamma'\sigma}) \to [D\sigma]^{s_2}_{\Gamma,\Gamma'\sigma,y:C\sigma}$	π-*cast* (6.2.16)	
	(10)	$([(y:C) \to D]_{\Gamma,x:A,\Gamma'})\,[\sigma] \equiv_{\Sigma_{\mathscr{C}}} (y:[C]_{\Gamma,x:A,\Gamma'}\,[\sigma]\,) \to [D]_{\Gamma,x:A,\Gamma',y:C}\,[\sigma]$	*Definition of* $[\cdot]$:	9
	(11)	$(y:[C]_{\Gamma,x:A,\Gamma'}\,[\sigma]\,) \to [D]_{\Gamma,x:A,\Gamma',y:C}\,[\sigma] \equiv_{\Sigma_{\mathscr{C}}} (y:[C\sigma]^{s_1}_{\Gamma,\Gamma'\sigma}) \to [D\sigma]^{s_2}_{\Gamma,\Gamma'\sigma,y:C\sigma}$	*Transitivity of* $\equiv_{\Sigma_{\mathscr{C}}}$	10,6,9
	(12)	$[C]_{\Gamma,x:A,\Gamma'}\,[\sigma] \equiv_{\Sigma_{\mathscr{C}}} [C\sigma]^{s_1\sigma}_{\Gamma,\Gamma'\sigma}$	*Lemma* (6.2.2)	11
	(13)	$[\![C]\!]_{\Gamma,x:A,\Gamma'}\,[\sigma] \equiv_{\Sigma_{\mathscr{C}}} [\![C\sigma]\!]_{\Gamma,\Gamma'\sigma}$	λ-*cast* (6.2.18)	12
	(14)	$([\lambda y:C.\, u]_{\Gamma,x:A,\Gamma'})\,[\sigma] \equiv_{\Sigma_{\mathscr{C}}} \lambda y:[\![C\sigma]\!]_{\Gamma,\Gamma'\sigma}.\,[u\sigma]^{D\sigma}_{\Gamma,\Gamma'\sigma}$	*Congruence of* $\equiv_{\Sigma_{\mathscr{C}}}$	7,13,5
	(15)	$\Gamma, \Gamma'\sigma, y:C\sigma \vdash_{\mathscr{C}} u\sigma \Leftarrow D\sigma$	*Substitution lemma* (6.2.6)	3,2
	(16)	$\lambda y:[\![C\sigma]\!]_{\Gamma,\Gamma'\sigma}.\,[u\sigma]^{D\sigma}_{\Gamma,\Gamma'\sigma} \equiv_{\Sigma_{\mathscr{C}}} [\lambda y:C\sigma.\, u\sigma]^{D\sigma}_{\Gamma,\Gamma'\sigma}$	*Definition of* $[\cdot]$:	15
	(17)	$([\lambda y:C.\, u]_{\Gamma,x:A,\Gamma'})\,[\sigma] \equiv_{\Sigma_{\mathscr{C}}} [\lambda y:C\sigma.\, u\sigma]^{(y:C\sigma)\to D\sigma}_{\Gamma,\Gamma'\sigma}$	*Transitivity of* $\equiv_{\Sigma_{\mathscr{C}}}$	14,16
	(18)	$([\lambda y:C.\, u]_{\Gamma,x:A,\Gamma'})\,[\sigma] \equiv_{\Sigma_{\mathscr{C}}} [(\lambda y:C.\, u)\,\sigma]^{((y:C)\to D)\sigma}_{\Gamma,\Gamma'\sigma}$	*Substitution*	17
★	(19)	$[M]_{\Gamma,x:A,\Gamma'}\,[\sigma] \equiv_{\Sigma_{\mathscr{C}}} [M\sigma]^{B\sigma}_{\Gamma,\Gamma'\sigma}$	*Definition of* M, B	18

◊ $\mathscr{C}^{\Rightarrow}_{app}$: $M = f\ a, B = D\{y \leftarrow a\}, \sigma := \{x \leftarrow N\}, [\sigma] := \{x \leftarrow [N]^{A}_{\Gamma}\}$

	(1)	$WS_{n+1}(\Gamma, x:A, \Gamma' \vdash_{\mathscr{C}} f\ a \Rightarrow D\{y \leftarrow a\})$	*Main hypothesis*	
	(2)	$\Gamma \vdash_{\mathscr{C}} N \Leftarrow A$		
	(3)	$WS_{n+1}(\Gamma, x:A, \Gamma' \vdash_{\mathscr{C}} f \Rightarrow (y:C) \to D)$	*Inversion on* $\mathscr{C}^{\Rightarrow}_{app}$	1
	(4)	$WS_{n+1}(\Gamma, x:A, \Gamma' \vdash_{\mathscr{C}} a \Leftarrow C)$		
	(5)	$[f]_{\Gamma,x:A,\Gamma'}\,[\sigma] \equiv_{\Sigma_{\mathscr{C}}} [f\sigma]^{((y:C)\to D)\sigma}_{\Gamma,\Gamma'\sigma}$	*Induction Hypothesis*	3,2
	(6)	$\Gamma, x:A, \Gamma' \vdash_{\mathscr{C}} a \Rightarrow C'$	*check-to-infer* (4.3.2)	4
	(7)	$C' \preceq_{\mathscr{C}} C$		
	(8)	$[a]^{C}_{\Gamma,x:A,\Gamma'}\,[\sigma] \equiv_{\Sigma_{\mathscr{C}}} [a\sigma]^{C\sigma}_{\Gamma,\Gamma'\sigma}$	*Induction Hypothesis*	4
	(9)	$[f]_{\Gamma,x:A,\Gamma'}\,[\sigma]\ \ [a]_{\Gamma,x:A,\Gamma'}\,[\sigma] \equiv_{\Sigma_{\mathscr{C}}} [f\sigma]^{((y:C)\to D)\sigma}_{\Gamma,\Gamma'\sigma}\ [a\sigma]^{C\sigma}_{\Gamma,\Gamma'\sigma}$	*Transitivity* $\equiv_{\Sigma_{\mathscr{C}}}$	
	(10)	$[f\ a]_{\Gamma,x:A,\Gamma'}\,[\sigma] = [f]_{\Gamma,x:A,\Gamma'}\,[\sigma]\ \ [a]^{C}_{\Gamma,x:A,\Gamma'}\,[\sigma]$	*Definition of* $[\cdot]$.	1
	(11)	$[f\sigma]^{(y:C\sigma)\to D\sigma}_{\Gamma,\Gamma'\sigma}\ [a\sigma]^{C\sigma}_{\Gamma,\Gamma'\sigma} \equiv_{\Sigma_{\mathscr{C}}} [f\sigma\ a\sigma]^{D\sigma}_{\Gamma,\Gamma'\sigma}$	*app-cast* (6.2.17)	1,3,6
	(12)	$[f\sigma\ a\sigma]^{D\sigma}_{\Gamma,\Gamma'\sigma} \equiv_{\Sigma_{\mathscr{C}}} [(f\ a)\,\sigma]^{D\sigma}_{\Gamma,\Gamma'\sigma}$	*substitution*	
	(13)	$[f\ a]_{\Gamma,x:A,\Gamma'}\,[\sigma] \equiv_{\Sigma_{\mathscr{C}}} [(f\ a)\,\sigma]^{D\sigma}_{\Gamma,\Gamma'\sigma}$	*Transitivity of* $\equiv_{\Sigma_{\mathscr{C}}}$	10,9,11,12
★	(14)	$[M]_{\Gamma,x:A,\Gamma'}\,\{x \leftarrow [N]^{A}_{\Gamma}\} \equiv_{\Sigma_{\mathscr{C}}} [M\sigma]^{B\sigma}_{\Gamma,\Gamma'\sigma}$		

◊ $\mathscr{C}^{\Rightarrow}_{\equiv}$:

	(1)	$WS_{n+1}(\Gamma, x:A, \Gamma' \vdash_{\mathscr{C}} M \Rightarrow B)$	*Main hypothesis*	
	(2)	$\Gamma \vdash_{\mathscr{C}} N \Leftarrow A$		
	(3)	$SUBST_n$		
	(4)	$WS_{n+1}(\Gamma, x:A, \Gamma' \vdash_{\mathscr{C}} M \Rightarrow C)$	*Inversion on* $\mathscr{C}^{\Rightarrow}_{\equiv}$	
	(5)	$WS_n(\Gamma, x:A, \Gamma' \vdash_{\mathscr{C}} B \Rightarrow s)$		
	(6)	$C \equiv_{\beta} B$		
	(7)	$[M]_{\Gamma,x:A,\Gamma'} \equiv_{\Sigma_{\mathscr{C}}} [M]^{C\sigma}_{\Gamma,\Gamma'\sigma}$	*Induction Hypothesis*	
	(8)	$CONV_n$	*Lemma* (6.2.22)	3
	(9)	$[M]^{C\sigma}_{\Gamma,\Gamma'\sigma} \equiv_{\Sigma_{\mathscr{C}}} [M]^{B\sigma}_{\Gamma,\Gamma'\sigma}$	$CONV_n$	8,1
★	(10)	$[M]_{\Gamma,x:A,\Gamma'} \equiv_{\Sigma_{\mathscr{C}}} [M]^{B\sigma}_{\Gamma,\Gamma'\sigma}$	*id-cast* (6.2.15)	9

◊ $\mathscr{C}^{\Rightarrow}_{\equiv^s}$: $B = s$

	(1)	$WS_{n+1}(\Gamma, x : A, \Gamma' \vdash_{\mathscr{C}} M \Rightarrow s)$	*Main hypothesis*	
	(2)	$\Gamma \vdash_{\mathscr{C}} N \Leftarrow A$		
	(3)	$SUBST_n$		
	(4)	$WS_{n+1}(\Gamma, x : A, \Gamma' \vdash_{\mathscr{C}} M \Rightarrow C)$	*Inversion on* $\mathscr{C}^{\Rightarrow}_{\equiv}$	
	(5)	$C \equiv_\beta s$		
	(6)	$WS_n(\Gamma, x : A, \Gamma' \vdash_{\mathscr{C}} C\ \mathbf{ws})$	*Inversion on* $\mathscr{C}^{\Rightarrow}_{\equiv}$	
	(7)	$[M]_{\Gamma, x:A, \Gamma'} \equiv_{\Sigma_{\mathscr{C}}} [M]^{C\sigma}_{\Gamma, \Gamma'\sigma}$	*Induction Hypothesis*	
	(8)	$CONV_n$	*Lemma (6.2.22)*	*3*
	(9)	$[M]_{\Gamma, x:A, \Gamma'} \equiv_{\Sigma_{\mathscr{C}}} [M]^{s}_{\Gamma, \Gamma'\sigma}$	$CONV_n$	*8,1*
★	(10)	$[M]_{\Gamma, x:A, \Gamma'} \equiv_{\Sigma_{\mathscr{C}}} [M]^{B\sigma}_{\Gamma, \Gamma'\sigma}$	*Definition of B*	*9*

◊ $\mathscr{C}^{\Leftarrow}_{\preceq}$:

Using the induction hypothesis, then it follows from the canonicity rule $\uparrow - \uparrow$.

◊ $\mathscr{C}^{\Leftarrow}_{\preceq_s}$:

Using the induction hypothesis, then it follows from the canonicity rule $\uparrow - \uparrow$

Lemma 6.2.25 *For all n, we have* $SUBST_n$.

Proof *By induction on n. The base case is handled with Lemma 6.2.23 and the inductive case with Lemma 6.2.24.*

Lemma 6.2.26 *If* $WS_n(\Gamma, x : A \vdash_{\mathscr{C}} B \Rightarrow s)$ *and* $\Gamma \vdash_{\mathscr{C}} N \Rightarrow A$ *then*

$$[\![B]\!]_{\Gamma, x:A} \{x \leftarrow [N]^{A}_{\Gamma}\} \equiv_{\Sigma_{\mathscr{C}}} [\![B\{x \leftarrow N\}]\!]_{\Gamma}$$

.

Proof *A consequence of the definition of* $[\![\cdot]\!]$. *and the substitution lemma 6.2.25.*

Lemma 6.2.27 *If* $WS_n(\Gamma \vdash_{\mathscr{C}} A \Rightarrow s), WS_n(\Gamma \vdash_{\mathscr{C}} B \Rightarrow s')$ *and* $A \equiv_\beta B$ *then there exists C such that* $A \hookrightarrow^*_\beta C \hookleftarrow^*_\beta B, [A]_\Gamma \equiv_{\Sigma_{\mathscr{C}}} [C]^{s}_{\Gamma}$ *and* $[B]_\Gamma \equiv_{\Sigma_{\mathscr{C}}} [C]^{s'}_{\Gamma}$

Proof *A consequence of the substitution lemma 6.2.25 and Lemma 6.2.21.*

Lemma 6.2.28 *If* $WS_n(\Gamma \vdash_{\mathscr{C}^{\Rightarrow}} A\ \mathbf{ws}), WS_n(\Gamma \vdash_{\mathscr{C}^{\Rightarrow}} B\ \mathbf{ws})$ *and* $A \equiv_\beta B$ *then* $[\![A]\!]_\Gamma \equiv_{\Sigma_{\mathscr{C}}} [\![B]\!]_\Gamma$.

Proof *A consequence of the definition or* $[\![\cdot]\!]$., *Lemma 6.2.27 and Lemma 6.2.13.*

6.2.4 Subtyping preservation

Lemma 6.2.29 *If* $WS_n(\Gamma \vdash_{\mathscr{C}} A \overset{?}{\Rightarrow} s_1), WS_n(\Gamma \vdash_{\mathscr{C}} B \overset{?}{\Rightarrow} s_2)$ *and* $A \preceq_{\mathscr{C}} B$ *then* $[A]_\Gamma\ \preceq^{s_2}_{s_1}\ [B]_\Gamma \equiv_{\Sigma^{Sp}_{\mathscr{C}}} \top$.

Proof *By induction on* $A \preceq^{t^-}_{\mathscr{C}} B$. *We recall that* $\preceq_{\mathscr{C}}$ *and* $\preceq^{t^-}_{\mathscr{C}}$ *define the same subtyping relation (see Lemma 1.7.18).*

For each case, we need to handle conversion with Lemma 6.2.28 (we recall that we use Assumption 1). Since conversion may introduce casts, we remove them using the canonicity rules $\mathrm{st} - \uparrow - l$ *and* $\mathrm{st} - \uparrow - r$.

◊ $\preceq^{r}_{\equiv_\beta}$:

Using the canonicity rule $\mathrm{st} - \equiv$.

◊ $\preceq^r_{\mathcal{C}_\mathscr{C}}$:
Using the canonicity rules $\mathrm{st} - s$.

◊ $\preceq^r_{\Pi}$:
Using the canonicity rule $\mathrm{st} - \pi$.

Typing preservation

At this stage, typing preservation could be proved by induction on the typing derivation. However we state some intermediate lemmas here which are purely computational to simplify the proof. All of them are straightforward.

Lemma 6.2.30 *If* $\Sigma_\mathscr{C}, \Gamma \vdash_\mathscr{D} [\![A]\!]_\Gamma : w$ *then* $w = \star$.

Proof *By definition of* $[\![\cdot]\!]$ ..

Lemma 6.2.31 **($[\![\cdot]\!]$. : $\star$)** *If* $\Sigma_\mathscr{C}, \Gamma \vdash_\mathscr{D} t : [\![A]\!]_\Gamma$ *then* $\Sigma_\mathscr{C}, \Gamma \vdash_\mathscr{D} [\![A]\!]_\Gamma : \star$.

Proof *By Lemma 5.1.3 and Lemma 6.2.30.*

Lemma 6.2.32 **($[\![\cdot]\!]$. $\leftrightarrow$ U)** *If* $\Sigma_\mathscr{C}, [\![\Gamma]\!] \vdash_\mathscr{D} t : [\![s]\!]_\Gamma$ *if and only if* $\Sigma_\mathscr{C}, [\![\Gamma]\!] \vdash_\mathscr{D} t : \mathbf{U}_s$

Proof *A consequence of the canonicity rule* $\mathbf{T} - s$ *(Assumption 1).*

Lemma 6.2.33 **($[\cdot]$. $\to$ $[\![\cdot]\!]$.)** *If* $\Gamma \vdash_\mathscr{C} A \Rightarrow s$ *and* $\Sigma_\mathscr{C}, [\![\Gamma]\!] \vdash_\mathscr{D} [A]_\Gamma : [\![s]\!]_\Gamma$ *then* $\Sigma_\mathscr{C}, [\![\Gamma]\!] \vdash_\mathscr{D} [\![A]\!]_\Gamma : \star$.

Proof *A consequence of the definition of* $[\![\cdot]\!]$ ..

Lemma 6.2.34 **(u : U)** *If* $(s, s') \in \mathcal{A}_\mathscr{C}$ *and* $\Sigma_\mathscr{C}, \Gamma \vdash_\mathscr{D} \mathbf{wf}$, *then* $\Sigma_\mathscr{C}, \Gamma \vdash_\mathscr{D} \mathbf{u}_{s,s'}\ \mathbf{I} : \mathbf{U}_{s'}$.

Proof *A consequence of signature specification* $\boldsymbol{\mathcal{A}}_{\Sigma^{Sp}_\mathscr{C}}$ *(Assumption 1).*

Lemma 6.2.35 **($[\![s]\!]$. : $\star$)** *If* $\Sigma_\mathscr{C}, [\![\Gamma]\!] \vdash_\mathscr{D} \mathbf{wf}$ *then* $\Sigma_\mathscr{C}, [\![\Gamma]\!] \vdash_\mathscr{D} [\![s]\!]_\Gamma : \star$

Proof *A consequence of the definition of* $[\![\cdot]\!]$ ..

Lemma 6.2.36 **($\boldsymbol{\pi}$: U)** *If* $(s_1, s_2, s) \in \mathcal{R}_\mathscr{C}$, $\Sigma_\mathscr{C}, \Gamma \vdash_\mathscr{D} t : \mathbf{U}_{s_1}$ *and* $\Sigma_\mathscr{C}, \Gamma \vdash_\mathscr{D} \lambda x : \mathbf{T}_{s_1}\ t.u : \mathbf{T}_{s_1}\ t \to \mathbf{U}_{s_2}$ *then* $\Sigma_\mathscr{C}, \Gamma \vdash_\mathscr{D} \boldsymbol{\pi}_{s_1,s_2,s}\ \mathbf{I}\ t\ (\lambda x : \mathbf{T}_{s_1}\ t.u) : \mathbf{U}_s$.

Proof *A consequence of signature specification* $\boldsymbol{\mathcal{R}}_{\Sigma^{Sp}_\mathscr{C}}$ *(Assumption 1).*

Lemma 6.2.37 **($[\![\cdot]\!]$. $\to$ $[\![\cdot]\!]$. : $\star$)** *If* $\Gamma \vdash_\mathscr{C} (x : A) \to B \Rightarrow s_3$, $\Sigma_\mathscr{C}, [\![\Gamma]\!] \vdash_\mathscr{D} [\![(x : A) \to B]\!]_\Gamma : \star$ *then* $\Sigma_\mathscr{C}, [\![\Gamma]\!] \vdash_\mathscr{D} (x : [\![A]\!]_\Gamma) \to [\![B]\!]_{\Gamma, x:A} : \star$.

Proof *By definition of* $[\![\cdot]\!]$. *using Lemma 6.2.14.*

Lemma 6.2.38 **($[\cdot]$. $\to$ $[\cdot]$:)** *If* $\Sigma^{Pu}_\mathscr{C} \models \Sigma^{Sp}_\mathscr{C}$, $A \vdash_\mathscr{C} s_1 \overset{?}{\Rightarrow}$, $B \vdash_\mathscr{C} s_2 \overset{?}{\Rightarrow}$, $\Sigma_\mathscr{C}, \Gamma \vdash_\mathscr{D} [A]_\Gamma : \mathbf{U}_{s_1}$, $\Sigma_\mathscr{C}, \Gamma \vdash_\mathscr{D} [B]_\Gamma : \mathbf{U}_{s_2}$, $\Sigma_\mathscr{C}, \Gamma \vdash_\mathscr{D} t : \mathbf{T}_{s_1}\ [A]_\Gamma$ *and* $A \preceq_\mathscr{C} B$ *then* $\Sigma_\mathscr{C}, \Gamma \vdash_\mathscr{D} {}^{s_2 \uparrow [B]_\Gamma}_{s_1 \uparrow [A]_\Gamma}\ \mathbf{I}\ t : \mathbf{T}_{s_2}\ [B]_\Gamma$.

Proof *A consequence of Lemma 6.2.29 and the signature specification* $\boldsymbol{\mathcal{C}}_{\Sigma^{Sp}_\mathscr{C}}$ *(Assumption 1).*

Definition 6.2.5 (WT)
We denote WT_n *the following property:*

- *If* $WS_n(\Gamma \vdash_{\mathscr{C}} t \Rightarrow A)$ *then* $\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [t]_\Gamma : [\![A]\!]_\Gamma$
- *If* $WS_n(\Gamma \vdash_{\mathscr{C}} t \Leftarrow A)$ *then* $\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [t]^A_\Gamma : [\![A]\!]_\Gamma$
- *If* $WS_n(\Gamma \vdash_{\mathscr{C}} \mathbf{wf})$ *then* $\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} \mathbf{wf}$

Lemma 6.2.39 (Soundness proof) *Assuming* WT_n *we have* WT_{n+1}.

Proof *By induction on the typing derivation.*

◊ $\mathscr{C}^{\Rightarrow \mathbf{wf}}_{\varnothing}$: $\Gamma = \varnothing$

	(1)	$[\![\varnothing]\!] = \varnothing$	*Definition of* $[\![\cdot]\!]$	
	(2)	$\Sigma_{\mathscr{C}} \vdash_{\mathscr{D}} \mathbf{wf}$	*Signature well-formed (6.2.7)*	
	(3)	$\Sigma_{\mathscr{C}}, \varnothing \vdash_{\mathscr{D}} \mathbf{wf}$	*Concatenation of typing context*	2
	(4)	$\Sigma_{\mathscr{C}}, [\![\varnothing]\!] \vdash_{\mathscr{D}} \mathbf{wf}$	*Congruence of equality*	3,1
★	(5)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} \mathbf{wf}$	*Definition of* Γ	4

◊ $\mathscr{C}^{\Rightarrow \mathbf{wf}}_{var}$: $\Gamma = \Gamma', x : A$

	(1)	$WS_{n+1}(\Gamma', x : A \vdash_{\mathscr{C}\Rightarrow} \mathbf{wf})$	*Main hypothesis*	
	(2)	$WS_{n+1}(\Gamma \vdash_{\mathscr{C}} A \Rightarrow s)$	*Inversion on* $\mathscr{C}^{\Rightarrow \mathbf{wf}}_{var}$	1
	(3)	$x \notin \Gamma$		
	(4)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [A]_\Gamma : [\![s]\!]_\Gamma$	*Induction Hypothesis*	2
	(5)	$x \notin \Sigma_{\mathscr{C}}$	$\mathcal{V} \cap \Sigma_{\mathscr{C}} = \emptyset$ *(Assumption 1)*	
	(6)	$x \notin [\![\Gamma]\!]$	*Lemma (6.2.8)*	3
	(7)	$x \notin \Sigma_{\mathscr{C}}, [\![\Gamma]\!]$	*Definition of* $\notin$	5,6
	(8)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [\![A]\!]_\Gamma : \star$	$[\cdot]\,. \to [\cdot]\,.$ *(6.2.33)*	4
★	(9)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!], x : [\![A]\!]_\Gamma \vdash_{\mathscr{D}} \mathbf{wf}$	$\mathscr{R}^{\mathbf{wf}}_{var}$	8,7

◊ $\mathscr{C}^{\Rightarrow}_{var}$: $t = x$

	(1)	$WS_{n+1}(\Gamma \vdash_{\mathscr{C}} x \Rightarrow A)$	*Main hypothesis*	
	(2)	$WS_n(\Gamma \vdash_{\mathscr{C}\Rightarrow} \mathbf{wf})$	*Inversion on* $\mathscr{C}^{\Rightarrow}_{var}$	1
	(3)	$x : A \in \Gamma$		
	(4)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} \mathbf{wf}$	*Induction Hypothesis*	2
	(5)	$x : [\![A]\!]_\Gamma \in [\![\Gamma]\!]$	*Lemma (6.2.8)*	3
	(6)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} x : [\![A]\!]_\Gamma$	$\mathscr{R}_{var}$	4,5
	(7)	$x = [x]_\Gamma$	*Definition of* $[\cdot]\,.$	
	(8)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [x]_\Gamma : [\![A]\!]_\Gamma$	*Congruence of equality*	6,7
★	(9)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [t]_\Gamma : [\![A]\!]_\Gamma$	*Definition of* t	8

◊ $\mathscr{C}^{\Rightarrow}_{sort}$: $t = s, A = s'$

	(1)	$WS_{n+1}(\Gamma \vdash_{\mathscr{C}} s \Rightarrow s')$	*Main hypothesis*	
	(2)	$WS_{n+1}(\Gamma \vdash_{\mathscr{C}\Rightarrow} \mathbf{wf})$	*Inversion on* $\mathscr{C}^{\Rightarrow}_{sort}$	1
	(3)	$(s, s') \mathcal{A}_{\mathscr{C}}$		
	(4)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} \mathbf{wf}$	*Induction Hypothesis*	2
	(5)	$\Sigma^{Pu}_{\mathscr{C}} \models \Sigma^{Sp}_{\mathscr{C}}$	*specif. sig. valid (Assumption 1)*	
	(6)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} \mathbf{u}_{s,s'}\ \mathbf{I} : \mathbf{U}_{s'}$	$\mathbf{u} : \mathbf{U}$ *(6.2.34)*	5,3, 4
	(7)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} \mathbf{u}_{s,s'}\ \mathbf{I} : [\![s']\!]_\Gamma$	$[\![\cdot]\!]\,. \leftrightarrow \mathbf{U}$ *(6.2.32)*	6
	(8)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [s]_\Gamma : [\![s']\!]_\Gamma$	*Definition of* $[\cdot]\,.$	7
★	(9)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [t]_\Gamma : [\![A]\!]_\Gamma$	*Definition of* t, A	8

◊ $\mathscr{C}^{\Rightarrow}_{\Pi}$: $t = (x:B) \to C, A = s$

	(1)	$WS_{n+1}(\Gamma \vdash_{\mathscr{C}} (x:B) \to C \Rightarrow s)$	*Main hypothesis*	
	(2)	$WS_{n+1}(\Gamma \vdash_{\mathscr{C}} B \Rightarrow s_1)$	*Inversion on* $\mathscr{C}^{\Rightarrow}_{\Pi}$	*1*
	(3)	$WS_{n+1}(\Gamma, x:B \vdash_{\mathscr{C}} C \Rightarrow s_2)$		
	(4)	$(s_1, s_2, s) \in \mathcal{R}_{\mathscr{C}}$		
	(5)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [B]_\Gamma : [\![s_1]\!]_\Gamma$	*Induction Hypothesis*	*2*
	(6)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [B]_\Gamma : \mathbf{U}_{s_1}$	$[\![\cdot]\!]_\cdot \leftrightarrow \mathbf{U}$ *(6.2.32)*	*5*
	(7)	$\Sigma_{\mathscr{C}}, [\![\Gamma, x:B]\!] \vdash_{\mathscr{D}} [C]_{\Gamma,x:B} : [\![s_2]\!]_{\Gamma,x:B}$	*Induction Hypothesis*	*3*
	(8)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!], x : [\![B]\!]_\Gamma \vdash_{\mathscr{D}} [C]_{\Gamma,x:B} : [\![s_2]\!]_{\Gamma,x:B}$	*Definition of* $[\![\cdot]\!]$	*7*
	(9)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!], x : [\![B]\!]_\Gamma \vdash_{\mathscr{D}} [C]_{\Gamma,x:B} : \mathbf{U}_{s_2}$	$[\![\cdot]\!]_\cdot \leftrightarrow \mathbf{U}$ *(6.2.32)*	*8*
	(10)	$(\star,\star,\star) \in \mathscr{D}$ *or* $(\star,\square,\square) \in \mathscr{D}$	*Definition of* $\mathscr{D}$	
	(11)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} \lambda x : [\![B]\!]_\Gamma . [C]_{\Gamma,x:B} : (x : [\![B]\!]_\Gamma) \to \mathbf{U}_{s_2}$	$\mathscr{R}_\lambda$	*9, 10*
	(12)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} \lambda x : [\![B]\!]_\Gamma . [C]_{\Gamma,x:B} : (x : \mathbf{T}_{s_1}\ [B]_\Gamma) \to \mathbf{U}_{s_2}$	*Definition of* $[\![\cdot]\!]_\cdot$	*11*
	(13)	$\Sigma^{Pu}_{\mathscr{C}} \models \Sigma^{Sp}_{\mathscr{C}}$	*specif. sig. valid (Assumption 1)*	
	(14)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} \boldsymbol{\pi}_{s_1,s_2,s}\ \mathbf{I}\ [B]_\Gamma\ (\lambda x : [\![B]\!]_\Gamma . [C]_{\Gamma,x:B}) : \mathbf{U}_s$	$\boldsymbol{\pi} : \mathbf{U}$ *(6.2.36)*	*13, 4, 6,12*
	(15)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [(x:B) \to C]_\Gamma : \mathbf{U}_s$	*Definition of* $[\cdot]_\cdot$	*1,14*
	(16)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [(x:B) \to C]_\Gamma : [\![s]\!]_\Gamma$	$[\![\cdot]\!]_\cdot \leftrightarrow \mathbf{U}$ *(6.2.32)*	*15*
★	(17)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [t]_\Gamma : [\![A]\!]_\Gamma$	*Definition of* t, A	*16*

◊ $\mathscr{C}^{\Rightarrow}_{\lambda}$: $t = \lambda x : B . u, A = (x:B) \to C$

	(1)	$WS_{n+1}(\Gamma \vdash_{\mathscr{C}} \lambda x : B . u \Rightarrow (x:B) \to C)$	*Main hypothesis*	
	(2)	$WS_{n+1}(\Gamma, x:B \vdash_{\mathscr{C}} u \Rightarrow C)$	*Inversion on* $\mathscr{C}^{\Rightarrow}_{\lambda}$	*1*
	(3)	$WS_n(\Gamma \vdash_{\mathscr{C}} (x:B) \to C \Rightarrow s)$		
	(4)	$\Gamma \vdash_{\mathscr{C}} B \Rightarrow s_1$	*Inversion prod (4.3.4)*	*3*
	(5)	$\Gamma, x:B \vdash_{\mathscr{C}} C \Rightarrow s_2$		
	(6)	$(s_1, s_2, s) \in \mathscr{C}$		
	(7)	$\Sigma_{\mathscr{C}}, [\![\Gamma, x:B]\!] \vdash_{\mathscr{D}} [u]_{\Gamma,x:B} : [\![C]\!]_{\Gamma,x:B}$	*Induction Hypothesis*	*2*
	(8)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!], x : [\![B]\!]_\Gamma \vdash_{\mathscr{D}} [u]_{\Gamma,x:B} : [\![C]\!]_{\Gamma,x:B}$	*Definition of* $[\![\cdot]\!]$	*7*
	(9)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [(x:B) \to C]_\Gamma : [\![s]\!]_\Gamma$	*Induction Hypothesis*	*3*
	(10)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [\![(x:B) \to C]\!]_\Gamma : \star$	$[\cdot]_\cdot \to [\![\cdot]\!]_\cdot$ *(6.2.33)*	*9*
	(11)	$(\star,\star,\star) \in \mathscr{D}$ *or* $(\star,\square,\square) \in \mathscr{D}$	*Definition of* $\mathscr{D}$	
	(12)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!], x : [\![B]\!]_\Gamma \vdash_{\mathscr{D}} \lambda x : [\![B]\!]_\Gamma . [u]_{\Gamma,x:B} : (x : [\![B]\!]_\Gamma) \to [\![C]\!]_{\Gamma,x:B}$	$\mathscr{R}_\lambda$	*8,11*
	(13)	$(x : \mathbf{T}_{s_1}\ [B]_\Gamma) \to \mathbf{T}_{s_2}\ [C]_{\Gamma,x:B \equiv_{\Sigma_{\mathscr{C}}}} \mathbf{T}_s\ (\boldsymbol{\pi}_{s_1,s_2,s}\ \mathbf{I}\ [B]_\Gamma\ (\lambda x . [C]_{\Gamma,x:B}))$	$\mathbf{T} - \pi$ *(6.4)*	
	(14)	$(x : [\![B]\!]_\Gamma) \to [\![C]\!]_{\Gamma,x:B \equiv_{\Sigma_{\mathscr{C}}}} [\![(x:B) \to C]\!]_\Gamma$	*Definition of* $[\![\cdot]\!]_\cdot, [\cdot]_\cdot$	*13, 4,5,6,3*
	(15)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [\![(x:B) \to C]\!]_\Gamma : \star$	$[\cdot]_\cdot \to [\![\cdot]\!]_\cdot$ *(6.2.33)*	*3,9*
	(16)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} \lambda x : [\![B]\!]_\Gamma . [u]_{\Gamma,x:B} : [\![(x:B) \to C]\!]_\Gamma$	$\mathscr{C}^{\Rightarrow}_{\equiv}$	*12,15,14*
	(17)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [\lambda x : B . u]_\Gamma : [\![(x:B) \to C]\!]_\Gamma$	*Definition* $[\cdot]_\cdot$	*16*
★	(18)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [t]_\Gamma : [\![A]\!]_\Gamma$	*Definition of* t, A	*17*

◊ $\mathscr{C}^{\Rightarrow}_{app}$: $t = f\ a, A = C\{x \leftarrow a\}$

(1)	$WS_{n+1}(\Gamma \vdash_{\mathscr{C}} f\ a \Rightarrow C\{x \leftarrow a\})$	*Main hypothesis*	
(2)	WT_n		
(3)	$WS_{n+1}(\Gamma \vdash_{\mathscr{C}} f \Rightarrow (x:B) \to C)$	*Inversion on* $\mathscr{C}^{\Rightarrow}_{app}$	*1*
(4)	$WS_{n+1}(\Gamma \vdash_{\mathscr{C}} a \Leftarrow B)$		
(5)	$\Gamma \vdash_{\mathscr{C}^{\Rightarrow}} (x:B) \to C$ **ws**	*Well-sorted* $\Rightarrow$ *(6.2.3)*	*3*
(6)	$WS_n(\Gamma \vdash_{\mathscr{C}} C\{x \leftarrow a\} \Rightarrow s_c)$	$WS_{\prec}$	*1*
(7)	$\Gamma \vdash_{\mathscr{C}} (x:B) \to C \Rightarrow s$	*Inversion on* $\mathbf{ws}^{\Rightarrow}$*-type*	*5*
(8)	$\Gamma \vdash_{\mathscr{C}} B \Rightarrow s_1$	*Inversion prod (4.3.4)*	*3*
(9)	$\Gamma, x:B \vdash_{\mathscr{C}} C \Rightarrow s_2$		
(10)	(s_1, s_2, s)		
(11)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [f]_\Gamma : [\![(x:B) \to C]\!]_\Gamma$	*Induction Hypothesis*	*3*
(12)	$(x:\mathbf{T}_{s_1}\ [B]_\Gamma) \to \mathbf{T}_{s_2}\ [C]_{\Gamma,x:B} \equiv_{\Sigma_{\mathscr{C}}} \mathbf{T}_s\ (\boldsymbol{\pi}_{s_1,s_2,s}\ \mathbf{I}\ [B]_\Gamma\ (\lambda x.\ [C]_{\Gamma,x:B}))$	$\mathbf{T}-\pi$ *(6.4)*	
(13)	$(x:[\![B]\!]_\Gamma) \to [\![C]\!]_{\Gamma,x:B} \equiv_{\Sigma_{\mathscr{C}}} [\![(x:B) \to C]\!]_\Gamma$	*Lemma (6.2.14)*	*7*
(14)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [\![(x:B) \to C]\!]_\Gamma : \star$	$[\![\cdot]\!]_\cdot : \star$ *(6.2.31)*	*7,17*
(15)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} (x:[\![B]\!]_\Gamma) \to [\![C]\!]_{\Gamma,x:B} : \star$	$[\![\cdot]\!]_\cdot \to [\![\cdot]\!]_\cdot : \star$ *(6.2.37)*	*14*
(16)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [f]_\Gamma : (x:[\![B]\!]_\Gamma) \to [\![C]\!]_{\Gamma,x:B}$	$\mathscr{R}_{\equiv_{\beta\Gamma}}$	*11,14,15*
(17)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [a]^B_\Gamma : [\![B]\!]_\Gamma$	*Induction Hypothesis*	*4*
(18)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [f]_\Gamma\ [a]^B_\Gamma : [\![C]\!]_\Gamma\{x \leftarrow [a]^B_\Gamma\}$	$\mathscr{R}_{app}$	*16,17*
(19)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [f]_\Gamma\ [a]^B_\Gamma : \mathbf{T}_{s_2}\ [C]_{\Gamma,x:B}\{x \leftarrow [a]^B_\Gamma\}$	*Definition of* $[\![\cdot]\!]_\cdot$	*18*
(20)	$\Gamma \vdash_{\mathscr{C}} C\{x \leftarrow B\} \Leftarrow s_2$	*Substitution lemma (6.2.6)*	*8,9*
(21)	$\Gamma \vdash_{\mathscr{C}} C\{x \leftarrow B\} \Rightarrow s_{2'}$	*check-to-infer (4.3.2)*	*20*
(22)	$s_{2'} \precsim_{\mathscr{C}} s_2$		
(23)	$[C]_{\Gamma,x:B}\{x \leftarrow [a]^B_\Gamma\} \equiv_{\Sigma_{\mathscr{C}}} [C\{x \leftarrow a\}]^B_\Gamma$	*Lemma (6.2.25)*	*9,4, 21*
(24)	$\mathbf{T}_{s_2}\ [C]_{\Gamma,x:B}\{x \leftarrow [a]^B_\Gamma\} \equiv_{\Sigma_{\mathscr{C}}} \mathbf{T}_{s_2}\ [C\{x \leftarrow a\}]^B_\Gamma$	*Congruence of* $\equiv_{\Sigma_{\mathscr{C}}}$	*23*
(25)	$\mathbf{T}_{s_2}\ [C\{x \leftarrow a\}]^B_\Gamma \equiv_{\Sigma_{\mathscr{C}}} \mathbf{T}_{s_{2'}}\ [C\{x \leftarrow a\}]_\Gamma$	$\mathbf{T}-\uparrow$ *(6.4)*	*24,21*
(26)	$\mathbf{T}_{s_2}\ [C]_{\Gamma,x:B}\{x \leftarrow [a]^B_\Gamma\} \equiv_{\Sigma_{\mathscr{C}}} \mathbf{T}_{s_{2'}}\ [C\{x \leftarrow a\}]_\Gamma$	*Transivity of* $\equiv_{\Sigma_{\mathscr{C}}}$	*24,25*
(27)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} \mathbf{T}_{s_{2'}}\ [C\{x \leftarrow a\}]_\Gamma : [\![s_c]\!]_\Gamma$	WT_n	*2,6*
(28)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} \mathbf{T}_{s_{2'}}\ [\![C\{x \leftarrow a\}]\!]_\Gamma : \star$	$[\cdot]_\cdot \to [\![\cdot]\!]_\cdot$ *(6.2.33)*	*27*
(29)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [f]_\Gamma\ [a]^B_\Gamma : \mathbf{T}_{s_2}\ [C\{x \leftarrow a\}]_\Gamma$	$\mathscr{R}_{\equiv_{\beta\Gamma}}$	*19,29,28*
(30)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [f]_\Gamma\ [a]^B_\Gamma : [\![C\{x \leftarrow a\}]\!]_\Gamma$	*Definition of* $[\![\cdot]\!]_\cdot$	*7,29*
(31)	$[f\ a]_\Gamma = [f]_\Gamma\ [a]^B_\Gamma$	*Definition of* $[\cdot]_\cdot$	*3,4*
(32)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [f\ a]_\Gamma : [\![C\{x \leftarrow a\}]\!]_\Gamma$	*Congruence of* $=$	*31,30*
✯ (33)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [t]_\Gamma : [\![A]\!]_\Gamma$	*Definition of* t, A	*32*

◊ $\mathscr{C}^{\Rightarrow}_{\equiv}$:

(1)	$WS_{n+1}(\Gamma \vdash_{\mathscr{C}} t \Rightarrow A)$	*Main hypothesis*	
(2)	$WS_{n+1}(\Gamma \vdash_{\mathscr{C}} t \Rightarrow B)$	*Inversion in* $\mathscr{C}^{\Rightarrow}_{\equiv}$	*1*
(3)	$WS_n(\Gamma \vdash_{\mathscr{C}} A \Rightarrow s)$		
(4)	$B \equiv_\beta A$		
(5)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [t]_\Gamma : [\![B]\!]_\Gamma$	*Induction Hypothesis*	*2*
(6)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [A]_\Gamma : [\![s]\!]_\Gamma$	*Induction Hypothesis*	*3*
(7)	$\Gamma \vdash_{\mathscr{C}^{\Rightarrow}} B$ **ws**	*Well-sorted* $\Rightarrow$ *(6.2.3)*	*2*
(8)	$\Gamma \vdash_{\mathscr{C}^{\Rightarrow}} A$ **ws**	*Well-sorted* $\Rightarrow$ *(6.2.3)*	*1*
(9)	$[\![B]\!]_\Gamma \equiv_{\Sigma_{\mathscr{C}}} [\![A]\!]_\Gamma$	*Lemma (6.2.28)*	*7,8,4*
(10)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [\![A]\!]_\Gamma : \star$	$[\cdot]_\cdot \to [\![\cdot]\!]_\cdot$ *(6.2.33)*	*3,6*
✯ (11)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [t]_\Gamma : [\![A]\!]_\Gamma$	$\mathscr{R}_{\equiv_{\beta\Gamma}}$	*5,10,9*

◊ $\mathscr{C}^{\Rightarrow}_{\equiv^s}$:

	(1)	$WS_{n+1}(\Gamma \vdash_{\mathscr{C}} t \Rightarrow s)$	*Main hypothesis*	
	(2)	$WS_n(\Gamma \vdash_{\mathscr{C}} t \Rightarrow B)$	*Inversion on* $\mathscr{C}^{\Rightarrow}_{\equiv s}$	*1*
	(3)	$B \equiv_\beta s$		
	(4)	$\Gamma \vdash_{\mathscr{C}^{\Rightarrow}} B$ **ws**	*Well-sorted* ⇒ *(6.2.3)*	*2*
	(5)	$\Gamma \vdash_{\mathscr{C}^{\Rightarrow}} s$ **ws**	*Well-sorted* ⇒ *(6.2.3)*	*1*
	(6)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [t]_\Gamma : [\![B]\!]_\Gamma$	*Induction Hypothesis*	*2*
	(7)	$[\![B]\!]_\Gamma \equiv_{\Sigma_{\mathscr{C}}} [\![s]\!]_\Gamma$	*Lemma (6.2.28)*	*4,5,3*
	(8)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}}$ **wf**	*Typing Context* **wf** *(5.1.2)*	*6*
	(9)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [\![s]\!]_\Gamma : \star$	$[\![s]\!]_\cdot : \star$ *(6.2.35)*	*8*
★	(10)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [t]_\Gamma : [\![s]\!]_\Gamma$	$\mathscr{R}_{\equiv_{\beta\Gamma}}$	*6,9,7*

◊ $\mathscr{C}^{\Leftarrow}_{\preceq}$:

	(1)	$WS_{n+1}(\Gamma \vdash_{\mathscr{C}} t \Leftarrow A)$	*Main hypothesis*	
	(2)	$WS_{n+1}(\Gamma \vdash_{\mathscr{C}} t \Rightarrow B)$	*Inversion on* $\mathscr{C}^{\Leftarrow}_{\preceq}$	*1*
	(3)	$WS_n(\Gamma \vdash_{\mathscr{C}} A \Rightarrow s)$		
	(4)	$B \preceq_{\mathscr{C}} A$		
	(5)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [t]_\Gamma : [\![B]\!]_\Gamma$	*Induction Hypothesis*	*2*
	(6)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [A]_\Gamma : [\![s]\!]_\Gamma$	*Induction Hypothesis*	*3*
	(7)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [\![B]\!]_\Gamma : \star$	$[\![\cdot]\!]_\cdot : \star$ *(6.2.31)*	*5*
	(8)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} {}^{s_2\uparrow^{[A]_\Gamma}}_{s_1\uparrow^{[B]_\Gamma}}\mathbf{I}\ [t]_\Gamma : \mathbf{T}_s\ [A]_\Gamma$	$[\cdot]_\cdot \to [\cdot]^{\cdot}_{\cdot}$ *(6.2.38)*	*6,7,4*
★	(9)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [t]^{A}_{\Gamma} : [\![A]\!]_\Gamma$	*definition of* $[\![\cdot]\!]_\cdot$ *and* $[\cdot]^{\cdot}_{\cdot}$	*8*

◊ $\mathscr{C}^{\Leftarrow}_{\preceq_s}$: $A = s$

	(1)	$WS_{n+1}(\Gamma \vdash_{\mathscr{C}} t \Leftarrow s)$	*Main hypothesis*	
	(2)	$WS_{n+1}(\Gamma \vdash_{\mathscr{C}} t \Rightarrow B)$	*Inversion on* $\mathscr{C}^{\Leftarrow}_{\preceq}$	*1*
	(3)	$B \preceq_{\mathscr{C}} s$		
	(4)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [t]_\Gamma : [\![B]\!]_\Gamma$	*Induction Hypothesis*	*2*
	(5)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}}$ **wf**	*Typing Context* **wf** *(5.1.2)*	*4*
	(6)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [\![s]\!]_\Gamma : \star$	$[\![s]\!]_\cdot : \star$ *(6.2.35)*	*5*
	(7)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [\![B]\!]_\Gamma : \star$	$[\![\cdot]\!]_\cdot : \star$ *(6.2.31)*	*4*
	(8)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} {}^{s_2\uparrow^{[s]_\Gamma}}_{s_1\uparrow^{[A]_\Gamma}}\mathbf{I}\ [t]_\Gamma : \mathbf{T}_{s'}\ [s]_\Gamma$	$[\cdot]_\cdot \to [\cdot]^{\cdot}_{\cdot}$ *(6.2.38)*	*6,7,3*
	(9)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [t]^{s}_{\Gamma} : [\![s]\!]_\Gamma$	*Definition of* $[\cdot]^{\cdot}_{\cdot}$ *and* $[\![\cdot]\!]_\cdot$	*8*
★	(10)	$\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [t]^{s}_{\Gamma} : [\![A]\!]_\Gamma$	*Definition of A*	*9*

Lemma 6.2.40 *We have* WT_0.

Proof *By inversion, there is no application at level* 0. *All the other cases can be handled the same way as in Lemma 6.2.39.*

Theorem 6.2.41 (Soundness of CTS encoding into the λΠ-CALCULUS MODULO THEORY) *Given a function specification* $\mathscr{C}$ *in normal form, a specification signature* $\Sigma^{Sp}_{\mathscr{C}}$ *and a private signature* $\Sigma^{Pr}_{\mathscr{C}}$ *if we have*

- *No clash between the symbol names of the encoding and variable names (Assumption 1)*
- *A valid specification signature:* $\Sigma^{Pu}_{\mathscr{C}} \models \Sigma^{Sp}_{\mathscr{C}}$ *(see Definition 6.1.5)*
- *A valid private signature:* $\Sigma^{Pu}_{\mathscr{C}} \models \Sigma^{Pr}_{\mathscr{C}}$ *(see Definition 6.1.6)*

then we have for all n,

- *If* $WS_n(\Gamma \vdash_{\mathscr{C}} t \Rightarrow A)$ *then* $\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [t]_\Gamma : [\![A]\!]_\Gamma$
- *If* $WS_n(\Gamma \vdash_{\mathscr{C}} t \Leftarrow A)$ *then* $\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [t]^A_\Gamma : [\![A]\!]_\Gamma$
- *If* $WS_n(\Gamma \vdash_{\mathscr{C}\Rightarrow} \mathbf{wf})$ *then* $\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} \mathbf{wf}$

Proof *By induction on* n. *The base case from Lemma 6.2.40 and the inductive case from Lemma 6.2.39.*

Using the equivalence between CTS and bi-directional CTS (see Theorem 4.3.9), one may deduce the soundness of the encoding for CTS:

Corollary 6.2.42 *Given a function specification* $\mathscr{C}$ *in normal form, a specification signature* $\Sigma^{Sp}_{\mathscr{C}}$ *and a private signature* $\Sigma^{Pr}_{\mathscr{C}}$ *if we have*

- *No clash between the symbol names of the encoding and variable names (Assumption 1)*
- *A valid specification signature:* $\Sigma^{Pu}_{\mathscr{C}} \models \Sigma^{Sp}_{\mathscr{C}}$ *(see Definition 6.1.5)*
- *A valid private signature:* $\Sigma^{Pu}_{\mathscr{C}} \models \Sigma^{Pr}_{\mathscr{C}}$ *(see Definition 6.1.6)*

then we have for all n,

- *If* $WS_n(\Gamma \vdash_{\mathscr{C}} t : A)$ *then* $\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} [t]^A_\Gamma : [\![A]\!]_\Gamma$
- *If* $WS_n(\Gamma \vdash_{\mathscr{C}} \mathbf{wf})$ *then* $\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} \mathbf{wf}$

where $[t]^A_\Gamma$ *(resp.* $[\![\Gamma]\!]$*) is the same encoding function pre-composed by the computable function extracted from Theorem 4.3.9 (the proof is constructive) that translate* $\Gamma \vdash_{\mathscr{C}} t : A$ *to* $\Gamma \vdash_{\mathscr{C}} t \Leftarrow A$ *(resp.* $\Gamma \vdash_{\mathscr{C}} \mathbf{wf}$ *to* $\Gamma \vdash_{\mathscr{C}\Rightarrow} \mathbf{wf}$*).*

6.3 Conservativity

We conjecture that this encoding is also conservative (see Definition 5.3.2).

Conjecture 13 (Conservativity of the encoding of CTS into the λΠ-calculus modulo theory) *If* $\Sigma_{\mathscr{C}}, [\![\Gamma]\!] \vdash_{\mathscr{D}} P : [\![A]\!]_\Gamma$ *then there exists* t *such that* $\Gamma \vdash_{\mathscr{C}} t \Leftarrow A$.

Conservativity is hard to prove because we need to reason on any term P. Maybe it would be possible to adapt the conservativity proof of Ali Assaf's for PTS [Ass15b]. As mentioned in Section 5.3.2, we are instead interested in to define a (partial) inverse function from the image of the encoding to the original CTS to prove that our encoding preserve the *shape* of a CTS term. However, from this inverse function we cannot prove the conservativity because conservativity needs to reason on all the terms, even the one which are not in the image of the translation.

Definition 6.3.1 (Inverse translation for CTS encoding)
We define the three partial inverse functions $|\cdot|$, $|\cdot|^{\uparrow}$ *and* $\|\cdot\|$ *by induction as follows*

$$
\begin{aligned}
|\mathbf{u}_{s,\cdot}\ \cdot| &:= s \\
|\boldsymbol{\pi}_{s_1,s_2,\cdot}\ \cdot\ \ \ | &:= \lambda\alpha : s_1.\, \lambda\beta : \alpha \to\ s_2.\, (x : \alpha) \to\ \beta\ x \\
|x| &:= x \\
|\lambda x : A.\, M| &:= \lambda x : \|A\|.\, |M| \\
|M\ N| &:= |M|\ |N|^{\uparrow}
\end{aligned}
$$

$$
\begin{aligned}
|\vdots\cdot\ \cdot\ A|^{\uparrow} &:= |A| \\
|A|^{\uparrow} &:= |A|
\end{aligned}
$$

$$
\begin{aligned}
\|\mathbf{U}_s\| &:= s \\
\|\mathbf{T}.\ A\| &:= |A|^{\uparrow} \\
\|(x : A) \to\ B\| &:= (x : \|A\|) \to\ \|B\| \\
\|\lambda x : A.\, M\| &:= \lambda x : \|A\|.\, \|B\| \\
\|M\ N\| &:= \|M\|\ |N|^{\uparrow}
\end{aligned}
$$

A well known issue is that the inverse translation for the code of a product generates unnecessary β redexes: The η reduction of a term such as $\boldsymbol{\pi}_{s_1,s_2,s_3}\ \mathbf{I}$ may be a valid term in the CTS encoding even if it is not in the image of the translation.

The functions defined above are actually right-inverse of the encoding functions modulo β conversion.

Lemma 6.3.1 *For every functional CTS specification in normal form,*

- *If* $\Gamma \vdash_{\mathscr{C}} t \Rightarrow A$ *then* $|[t]_{\Gamma}| \equiv_{\beta} t$.
- *If* $\Gamma \vdash_{\mathscr{C}} t \Leftarrow A$ *then* $\left|[t]^{A}_{\Gamma}\right|^{\uparrow} \equiv_{\beta} t$.
- *If* $\Gamma \vdash_{\mathscr{C}\Rightarrow} A$ **ws** *then* $\|[\![A]\!]_{\Gamma}\| \equiv_{\beta} A$.

Proof *By a mutual induction on the term* t *and* A *for the last one.*

These inverse functions ensure that our encoding functions are not trivial, but do not ensure that the encoding is conservative. Below we show why using a lift operator which acts on sorts only (as proposed by Ali Assaf [Ass15b]) is not conservative.

Example 6.6 *The lift operator can be applied only on sorts and not types. Hence, to simulate subtyping over products, we sometimes need to eta-expand some terms. Going back the specification of* Lean*(Definition 1.5.13) and given the following typing context* Γ*:*

- $P : \mathfrak{0} \to \mathfrak{0}$
- $eq : (\mathfrak{0} \to \mathfrak{3}) \to (\mathfrak{0} \to \mathfrak{3}) \to \mathfrak{0}$
- $refl : (x : \mathfrak{0} \to \mathfrak{3}) \to eq\ x\ x$

Then we can derive the judgment $\Gamma \vdash_{\mathscr{C}^{\Rightarrow}} eq\ P\ (\lambda x : \mathfrak{0}.\ P\ x)$ **ws**. *However, it is easy to see that there is no term* t *such that* $\Gamma \vdash_{\mathscr{C}} t \Leftarrow eq\ P\ (\lambda x : \mathfrak{0}.\ P\ x)$. *Indeed, the specification is consistent and terminating, therefore this type should be inhabited by term in normal form, and the only possibility for that is to use the constant* $refl$. *But since* P *and* $\lambda x : \mathfrak{0}.\ P\ x$ *are not convertible this is not possible. However, if we encode this judgment the lift operator, we need to eta-expand the variable* P *to lift the type* $\mathfrak{0} \to \mathfrak{0}$ *to* $\mathfrak{0} \to \mathfrak{2}$. *Therefore, the translated type becomes the same as if we had translated the type* $eq\ (\lambda x : \mathfrak{0}.\ P\ x)\ (\lambda x : \mathfrak{0}.\ P\ x)$ *which is inhabited by the translation of* $refl\ (\lambda x : \mathfrak{0}.\ P\ x)$. *Hence conservativity if broken since there is a type in the original system which is not inhabited but its translation is inhabited.*

6.4 Future Work

Encoding functions over a derivation tree: We have preferred in this work to have encoding functions on judgments instead of derivation trees. However, expressing these functions on derivation trees would avoid using bi-directional CTS. Since the equivalence between CTS and bi-directional CTS is only for CTS specification in normal form, it might be interesting to see whether the encoding expressed on derivation trees allows the embedding of a larger class of CTS specifications.

Reformulating the proof with an explicit conversion: Instead of using the well-structured hypothesis, it might be interesting to have a proof using an explicitly typed conversion. This way, the proof relies on a weaker property than well-structured derivation trees since we have showed in Section 3.3 that well-structured derivation trees implies the equivalence between untyped and typed conversion.

Identity casts: While the identity cast is used in the soundness proof, we conjecture that we could prove the soundness of our encoding without it. We used the identity cast in the proof of lemma 6.2.25 which shows the encoding functions permute with substitutions. However, we think that this lemma could be weakened so that there is no need for the identity cast rule $\uparrow - \mathrm{id}$ (6.4). To do so, we propose to weaken the substitution lemma as follows:

$$\text{If } \Gamma, x : A, \Gamma' \vdash_{\mathscr{C}} t \Leftarrow B \text{ and } \Gamma \vdash_{\mathscr{C}} N \Leftarrow A \text{ then}$$

$$[M]^{B}_{\Gamma, x:A, \Gamma'} \left\{ x \leftarrow [N]^{A}_{\Gamma} \right\} \equiv_{\Sigma_{\mathscr{C}}} [M\ \{x \leftarrow N\}]^{B\{x \leftarrow N\}}_{\Gamma, \Gamma'\{x \leftarrow N\}}$$

To be used, this require to change the translation function $[\![\cdot]\!]$. by the following one instead:

$$[\![A]\!]_{\Gamma} = \mathbf{T}_s\ [A]^{s}_{\Gamma}$$
$$\text{when } \Gamma \vdash_{\mathscr{C}} A \overset{?}{\Rightarrow} s$$

Another advantage is that identity casts are the only reason that we needed the well-structured derivation tree hypothesis (see 6.2.22). Hence by removing the identity cast canonicity rule, there might be a chance that the well-structured hypothesis is not required anymore.

However, identity casts are useful in practice as argued in Section 6.1.4. Moreover, we observed empirically that when identity casts are mixed with inductive types (presented in Chapter 8), identity casts become mandatory otherwise type checking fails. However, we have no theoretical justification for this since we did not investigate thoroughly the encoding of inductive types into the λΠ-CALCULUS MODULO THEORY nor DEDUKTI.

Remove the well-structured hypothesis in the typing preservation proof: The only reason why we need to do an induction over the level while showing the preservation of typing was because in the application case $\mathscr{C}^{\Rightarrow}_{app}$ we were lacking the hypothesis that that $\Gamma \vdash_{\mathscr{C}} B\{x \leftarrow a\} \Rightarrow s$. We have shown in Section 3.4.2 that this hypothesis can be safely added as a premise of the application rule for CTS $\mathscr{C}_{app}$ without changing the expressivity of the typing system. By equivalence, this premise also could be added for the rule $\mathscr{C}^{\Rightarrow}_{app}$. Hence, combining this remark with the remark about the identity cast above, it may be possible to prove the embedding without the well-structured hypothesis.

Toward a formalization of the proof In this chapter, we have tried to present a readable proofs which had the disadvantage to leave out some details. To be really confident about the proof, it would be interesting to formalize these proofs in a proof assistant. We also realized that while translating judgments helps to get readable proofs, it adds a lot of complexity to fillin all the tiny details because the price of this was to use partial functions.

Moreover, this translation shows that the cast operator as a computational content. Ali Assaf already showed the computational content for the lift operator [Ass14] for the calculus of construction. Probably, in a similar way, we could define a CTS with an explicit cast. We conjecture that the 8 canonical equalities (Definition 6.4) used for this translation would be needed to prove the equivalence between a CTS with an explicit cast with CTS using an implicit subtyping. This type system may be useful per se, but it also interesting since it would give a nice way to express our translation functions to the λΠ-CALCULUS MODULO THEORY on terms. This way we would have total functions which are easier to deal with a proof assistant.

Chapter 7

STT∀: A Constructive Version of Higher-Order Logic

This chapter presents the STT∀ logic, an extension of Simple Type Theory [Far08] with prenex polymorphism and type operators. This logic is powerful enough to express arithmetic theorems easily, but weak enough so that it is easy to export theorems from this logic to several other systems, making this logic suitable for interoperability. STT∀ has been implemented in the logical framework λΠ-calculus modulo theory and its implementation in Dedukti is presented in Chapter 8. We illustrate its adequacy for exporting theorems in Chapter 12 by showing how proofs in STT∀ can be exported to Coq, Matita, Lean, PVS and OpenTheory, the latter being used to target proof systems based on HOL (Higher-Order Logic).

The restriction of STT∀ to prenex polymorphism is a consequence of the logical inconsistency of the system U^- (Definition 1.5.10) [Hur95]. This remark will be detailed in Section 7.2, where we show that STT∀ can be embedded into a CTS. This embedding also gives us a proof of its consistency for free, because there is a sort-morphism from the CTS specification of STT∀ to a consistent CTS. However, the CTS encoding is a little bit heavy to use in practice. We will see a light implementation of the embedding of STT∀ into Dedukti in Chapter 8.

7.1 Definition of STT∀

STT∀ is an intutionistic version of Simple Type Theory [Far08] with prenex polymorphism and type operators. In this work, we formulate Simple Type Theory (also known as Church's Type Theory) as the PTS λHOL (as described in Chapter 1) with a type variable $\iota : \Box$ to represent the type of natural numbers. One drawback of Simple Type Theory is its lack of polymorphism which makes this system inefficient to use in practice. For example, without polymorphism, there is one equality symbol for each needed type. This leads to proving the reflexivity of equality for each equality symbol, while it is the same proof for every type. Jean-Yves Girard [Gir72] [Coq86] proved that adding full polymorphism[1] (such as System F) to Simple Type Theory makes this system logically inconsistent. This paradox has also been formulated on PTS and simplified by Hurkens [Hur95] which gave the PTS specification U^-.

It is known that, in order to avoid such inconsistency, the polymorphism could be restricted to prenex polymorphism. Prenex polymorphism extends Simple Type Theory in a consistent way because every polymorphic type can be made *monomorphic* by instantiation and duplication.

[1] Coquand's paper [Coq86] also shows that omitting types annotations for polymorphic types would make the logic inconsistent.

$$
\begin{array}{lccll}
\text{Types} & A, B & \in \mathcal{T} & ::= & X \mid \mathbf{prop} \mid A \to B \mid p\ A_1\ \ldots\ A_n \\
\text{PolyTypes} & T & \in \mathcal{T} & ::= & A \mid \forall X.\ T \\
\text{Terms} & t, u & \in \mathcal{T} & ::= & x \mid \lambda x : T.\ t \mid \lambda X.\ t \mid t\ u \mid t\ A \mid \\
 & & & & t \Rightarrow u \mid \forall x : A.\ t \mid \forall X.\ t \\
\text{Contexts} & \Gamma & \in \mathcal{G} & ::= & \varnothing \mid \Gamma, x : A \mid \Gamma, X \mid \Gamma, (p, n) \\
\text{Hypothesis} & \Xi & \in \mathcal{G} & ::= & \varnothing \mid \Xi, t
\end{array}
$$

Figure 7.1: STT∀ syntax

This means that any derivation using prenex polymorphism can be translated back to Simple Type Theory.

But prenex polymorphism is not enough to express type operators such as `list`. A type operator is constructed using a name and an arity. For example, `list` is a type operator of arity 1 because it takes one type as parameter.

These are the two main features of STT∀ over Simple Type Theory. The STT∀ syntax is presented in Fig. 7.1. The types for propositions **prop** and functions $\to$, could be declared as type operators, of arity 0 and 2 respectively. Since they have a particular meaning for the typing judgment, we add them explicitly to the syntax. Also, STT∀ allows the declaration and the definition of constants. Declaring constants is great for interoperability because the user in the target system is free to use the definition he wants. However, it also means that this operator comes with properties (or axioms) that needs to be proven by the user. The typing system and the proof system are presented in Fig. 7.2 and Fig. 7.3. Finally, we point out that, in STT∀, two terms are considered equal if they are convertible up to β and δ (unfolding of constants).

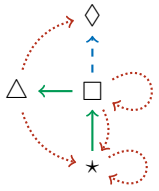
In the next section, we explore the logic STT∀ from the point of view of CTS. This will give an easy proof of its consistency and also allows us to understand possible extensions of STT∀.

7.2 STT∀ and CTS

In this section, we explore how STT∀ can be seen as a CTS. Since STT∀ extends Simple Type Theory with prenex polymorphism, its associated CTS is an extension of λHOL as defined by Geuvers [Geu93]. Thanks to the cumulativity relation, it is possible to express prenex polymorphism. This is pictured in the definition of the CTS STT∀$^-$.

Definition 7.2.1 (STT∀$^-$)

$$
(STT\forall^-) = \begin{cases} \mathcal{S} = & \{\star, \Box, \triangle, \Diamond\} \\ \mathcal{A} = & \{(\star, \Box), (\Box, \triangle)\} \\ \mathcal{R} = & \{(\star,\star,\star), (\Box,\Box,\Box), (\Box,\star,\star), (\triangle,\Diamond,\Diamond), (\triangle,\star,\star)\} \\ \mathcal{C} = & \{\Box, \Diamond\} \end{cases}
$$



With respect to λHOL, we introduce a new sort $\Diamond$ which is not the type of $\Box$, instead $\Box$ is a subtype of $\Diamond$. The idea behind this design is to see the sort $\Box$ as the sort for monomorphic types while $\Diamond$ is the sort for polymorphic types. The cumulativity relation between $\Box$ and $\Diamond$ represents that any monomorphic type is also a polymorphic type. From STT∀$^-$, there exist two interesting sort-morphisms:

$$\frac{}{\varnothing \vdash_{\mathscr{J}} \mathbf{wf}}\ \mathscr{J}_{\varnothing}^{\mathbf{wf}} \qquad \frac{\Gamma \vdash_{\mathscr{J}} \mathbf{wf} \qquad X \notin \Gamma}{\Gamma, X \vdash_{\mathscr{J}} \mathbf{wf}}\ \mathscr{J}_{tvar}^{\mathbf{wf}}$$

$$\frac{\Gamma \vdash_{\mathscr{J}} \mathbf{wf} \qquad p \notin \Gamma}{\Gamma, (p, n) \vdash_{\mathscr{J}} \mathbf{wf}}\ \mathscr{J}_{tyop}^{\mathbf{wf}}$$

$$\frac{\Gamma \vdash_{\mathscr{J}} \mathbf{wf} \qquad x \notin \Gamma \qquad \Gamma \vdash_{\mathscr{J}} A : \mathbf{type}}{\Gamma, x : A \vdash_{\mathscr{J}} \mathbf{wf}}\ \mathscr{J}_{var}^{\mathbf{wf}}$$

$$\frac{\Gamma \vdash_{\mathscr{J}} \mathbf{wf}}{\Gamma \vdash_{\mathscr{J}} \mathbf{prop} : \mathbf{type}}\ \mathscr{J}_{\mathbf{prop}}^{\mathbf{type}} \qquad \frac{\Gamma \vdash_{\mathscr{J}} \mathbf{wf} \qquad X \in \Gamma}{\Gamma \vdash_{\mathscr{J}} X : \mathbf{type}}\ \mathscr{J}_{tvar}^{\mathbf{type}}$$

$$\frac{\Gamma \vdash_{\mathscr{J}} A : \mathbf{type} \qquad \Gamma \vdash_{\mathscr{J}} B : \mathbf{type}}{\Gamma \vdash_{\mathscr{J}} A \to B : \mathbf{type}}\ \mathscr{J}_{\to}^{\mathbf{type}}$$

$$\frac{\Gamma \vdash_{\mathscr{J}} A_i : \mathbf{type} \qquad (p, n) \in \Gamma}{\Gamma \vdash_{\mathscr{J}} p\ A_1\ \ldots\ A_n : \mathbf{type}}\ \mathscr{J}_{tyop}^{\mathbf{type}}$$

$$\frac{\Gamma, X \vdash_{\mathscr{J}} T : \mathbf{type} \qquad X \notin \Gamma}{\Gamma \vdash_{\mathscr{J}} \forall X.\ T : \mathbf{type}}\ \mathscr{J}_{\forall}^{\mathbf{type}}$$

$$\frac{\Gamma \vdash_{\mathscr{J}} \mathbf{wf} \qquad (x, A) \in \Gamma}{\Gamma, x : A \vdash_{\mathscr{J}} x : A}\ \mathscr{J}_{var}$$

$$\frac{\Gamma, x : A \vdash_{\mathscr{J}} t : B}{\Gamma \vdash_{\mathscr{J}} \lambda x : A.\, t : A \to B}\ \mathscr{J}_{\lambda}$$

$$\frac{\Gamma \vdash_{\mathscr{J}} t : A \to B \qquad \Gamma \vdash_{\mathscr{J}} u : A}{\Gamma \vdash_{\mathscr{J}} t\ u : B}\ \mathscr{J}_{app}$$

$$\frac{\Gamma \vdash_{\mathscr{J}} t : \mathbf{prop} \qquad \Gamma \vdash_{\mathscr{J}} u : \mathbf{prop}}{\Gamma \vdash_{\mathscr{J}} t \Rightarrow u : \mathbf{prop}}\ \mathscr{J}_{\Rightarrow}$$

$$\frac{\Gamma, x : A \vdash_{\mathscr{J}} t : \mathbf{prop}}{\Gamma \vdash_{\mathscr{J}} \forall x : A.\ t : \mathbf{prop}}\ \mathscr{J}_{\forall}$$

$$\frac{\Gamma, X \vdash_{\mathscr{J}} t : \mathbf{prop}}{\Gamma \vdash_{\mathscr{J}} \forall X.\ t : \mathbf{prop}}\ \mathscr{J}_{\forall}$$

$$\frac{\Gamma, X \vdash_{\mathscr{J}} t : T}{\Gamma \vdash_{\mathscr{J}} \lambda X.\, t : \forall X.\ T}\ \mathscr{J}_{\lambda^T}$$

$$\frac{\Gamma \vdash_{\mathscr{J}} t : \forall X.\ T \qquad \Gamma \vdash_{\mathscr{J}} A : \mathbf{type}}{\Gamma \vdash_{\mathscr{J}} t\ A : T\,\{X \leftarrow A\}}\ \mathscr{J}_{app^T}$$

Figure 7.2: STT∀ Typing System

$$\frac{\Gamma \vdash_{\mathscr{S}} t : \mathbf{prop}}{\Gamma;\Xi, t \vdash_{\mathscr{S}} t}\ \mathscr{S}_{var} \qquad \frac{\Gamma;\Xi \vdash_{\mathscr{S}} t \qquad t \equiv_\beta u}{\Gamma;\Xi \vdash_{\mathscr{S}} u}\ \mathscr{S}_{\equiv_\beta}$$

$$\frac{\Gamma;\Xi, t \vdash_{\mathscr{S}} u}{\Gamma;\Xi \vdash_{\mathscr{S}} t \Rightarrow u}\ \mathscr{S}_{\Rightarrow_I} \qquad \frac{\Gamma;\Xi \vdash_{\mathscr{S}} t \Rightarrow u \qquad \Gamma;\Xi \vdash_{\mathscr{S}} t}{\Gamma;\Xi \vdash_{\mathscr{S}} u}\ \mathscr{S}_{\Rightarrow_E}$$

$$\frac{\Gamma, x : A;\Xi \qquad x \notin \Gamma}{\Gamma;\Xi \vdash_{\mathscr{S}} \forall x : A.\ t}\ \mathscr{S}_{\forall_I} \qquad \frac{\Gamma;\Xi \vdash_{\mathscr{S}} \forall x : A.\ t \qquad \Gamma \vdash_{\mathscr{S}} u : A}{\Gamma;\Xi \vdash_{\mathscr{S}} t\{x \leftarrow u\}}\ \mathscr{S}_{\forall_E}$$

$$\frac{\Gamma, X;\Xi \vdash_{\mathscr{S}} t \qquad X \notin \Gamma}{\Gamma;\Xi \vdash_{\mathscr{S}} \forall X.\ t}\ \mathscr{S}_{\forall_I} \qquad \frac{\Gamma;\Xi \vdash_{\mathscr{S}} \forall X.\ t \qquad \Gamma, A \vdash_{\mathscr{S}} \mathbf{wf}}{\Gamma;\Xi \vdash_{\mathscr{S}} t\{X \leftarrow A\}}\ \mathscr{S}_{\forall_E}$$

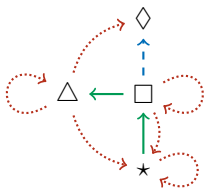
Figure 7.3: STT∀ Proof System

- The first one is a sort-morphism from STT∀$^-$ to $\mathscr{C}_3$ as proved in Theorem 7.2.1 which implies that STT∀$^-$ is logically consistent (β terminates): If the type false, $(x : \star) \to x$ is inhabited, it is inhabited by a term in normal form using β. But it is easy to see that such term does not exists.
- The second one is a sort-morphism from STT∀$^-$ to U^- where the sorts $\Diamond$ and $\Box$ are merged which emphasizes that polymorphism of STT∀$^-$ is weaker than full polymorphism.

However, the CTS STT∀$^-$ is not a faithful representation of STT∀ because it cannot represent a type operator. Indeed, a type operator such as *list* should have the type (in the CTS) $\Box \to \Box$ which is not possible in STT∀$^-$. This requires to add a new product (a new rule in the specification). One way would be to add the rule $(\triangle, \triangle, \triangle)$.

Definition 7.2.2 (STT∀$^+$)

$$(STT\forall^+) = \begin{cases} \mathcal{S} = & \{\star, \Box, \triangle, \Diamond\} \\ \mathcal{A} = & \{(\star, \Box), (\Box, \triangle)\} \\ \mathcal{R} = & \{(\star,\star,\star), (\Box,\Box,\Box), (\Box,\star,\star), (\triangle,\Diamond,\Diamond) \\ & , (\triangle,\star,\star), (\triangle,\triangle,\triangle)\} \\ \mathcal{C} = & \{\Box, \Diamond\} \end{cases}$$



This new specification is called STT∀$^+$ because this rule is more expressive than what can be achieved in STT∀. This is because STT∀$^+$ allows to use type variables parameterized by other type variables. For example, one can write in STT∀$^+$ the judgment

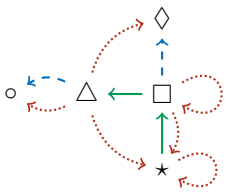
$$\vdash_{\text{STT}\forall^+} (M : (\Box \to \Box)) \to (A : \Box) \to M\ A \to M\ A : \Diamond$$

which is not possible in STT∀. Indeed, the type of a type variable X is STT∀ can only be $\Box$. This is the same as saying that the sort $\triangle$ has only one inhabitant which is $\Box$.

However, the rule added in STT$\forall^+$ provides new inhabitants for the sorts $\triangle$ which do not exist in STT$\forall^-$. Hence, to provide a faithful representation of STT$\forall$ as a CTS, we add a new sort $\circ$ and the product $(\triangle, \triangle, \circ)$ instead.

Definition 7.2.3 (STT∀)

$$(STT\forall) = \begin{cases} \mathcal{S} = & \{\star, \Box, \triangle, \Diamond, \circ\} \\ \mathcal{A} = & \{(\star, \Box), (\Box, \triangle)\} \\ \mathcal{R} = & \{(\star,\star,\star), (\Box,\Box,\Box), (\Box,\star,\star), (\Box,\Diamond,\Diamond) \\ & , (\Diamond,\star,\star), (\triangle,\circ,\circ)\} \\ \mathcal{C} = & \{(\Box,\Diamond), (\triangle,\circ)\} \end{cases}$$



The logical consistency of STT$\forall^-$, STT$\forall^+$, and STT$\forall$ is hence ensured since there is a sort-morphism from these specifications to the CTS of Coq which is terminating [Wer94].

Theorem 7.2.1 (Consistency of STT∀) *The CTS specifications STT$\forall^-$, STT$\forall^+$ and STT$\forall$ are logically consistent.*

Proof *There is a sort-morphism from these specifications to $\mathscr{C}_4$ (as defined in Definition 1.5.12):*

- $\star$ *is mapped to* $\mathbf{0}$
- $\Box$ *is mapped to* $\mathbf{1}$
- $\triangle$, $\Diamond$ *and* $\circ$ *are mapped to* $\mathbf{2}$

We can conclude with Theorem 2.2.2 and Theorem 2.2.3.

7.2.1 From STT∀ to CTS

One can prove that the first representation of STT$\forall$ can be embedded into a CTS. This translation is given in Fig. 7.4. Notice that this translation goes from a judgment to a judgment.

Lemma 7.2.2

- *If* $\Gamma \vdash_{\mathscr{S}}$ **wf** *then* $[\Gamma] \vdash_{STT\forall}$ **wf**
- *If* $\Gamma \vdash_{\mathscr{S}} A :$ **type** *then* $[\Gamma] \vdash_{STT\forall} [A] : \Box$
- *If* $\Gamma \vdash_{\mathscr{S}} t : A$ *then* $[\Gamma] \vdash_{STT\forall} [t] : [A]$
- *If* $\Gamma \vdash_{\mathscr{S}} t : T$ *then* $[\Gamma] \vdash_{STT\forall} [t] : [T]$
- *If* $\Gamma; \Xi \vdash_{\mathscr{S}} t$ *then there exists* p *such that* $[\Gamma], [\Xi] \vdash_{STT\forall} p : [t]$ *and* $[\Gamma] \vdash_{STT\forall} [t] : \star$

Proof *A straight induction on the derivation tree.*

We conjecture that actually these two representations are equivalent. This conjecture will be explored in Section 9.3.1 on the Dedukti side using the tool Dkmeta presented in Chapter 9.

$$
\begin{aligned}
[X] &= X \\
[\mathbf{prop}] &= \star \\
[A \to B] &= [A] \to [B] \\
[p\ A_1\ \ldots\ A_n] &= p\ [A_1]\ \ldots\ [A_n]
\end{aligned}
$$

$$
\begin{aligned}
[A] &= [A] \\
[\forall X.\ T] &= (X : \Box) \to [T]
\end{aligned}
$$

$$
\begin{aligned}
[x] &= x \\
[\lambda x : T.\, t] &= \lambda x : [T].\, [t] \\
[\lambda X.\, t] &= \lambda X.\, [t] \\
[t\ u] &= [t]\ [u] \\
[t\ u] &= [t]\ [A] \\
[t \Rightarrow u] &= [t] \Rightarrow [u] \\
[\forall x : A.\ t] &= (x : [A]) \to [t] \\
[\forall X.\ t] &= (x : \Box) \to [t]
\end{aligned}
$$

$$
\begin{aligned}
[\varnothing] &= \varnothing \\
[\Gamma, x : A] &= x : [A], [\Gamma] \\
[\Gamma, X] &= X : \Box, [\Gamma] \\
[\Gamma, (p, n)] &= p : \underbrace{\Box \to \cdots \to \Box}_{n \text{ times}} \to \Box, [\Gamma]
\end{aligned}
$$

Figure 7.4: A translation of STT∀ to CTS

7.3 Translation into λΠ-calculus modulo theory

Given the embedding from STT∀ to its CTS representation, the translation in λΠ-calculus modulo theory can be done using the embedding of CTS in Dedukti as seen in Chapter 6 since the CTS is functional and in normal form. This allows us to derive automatically the correction of this translation. However, the original formalization of STT∀ in Dedukti was not done from a formulation of STT∀ as a CTS but rather from the original formulation of it, as presented in the beginning of this chapter. This original formalization will be detailed for Dedukti in Chapter 8.

7.4 Future work

Equivalence between STT∀ and its CTS representation: We have shown that the first formulation could be encoded as a CTS. However, we conjecture that any derivation tree in the CTS representation could be translated as a derivation tree in STT∀. This should not be difficult; however, it requires a little bit of work because, as for implicit subtyping, it is not clear from the CTS judgment which product needs to be used: one needs to look at the derivation tree. However, this inverse translation can be expressed easily in Dedukti as a translation from judgment to judgment since the encoding of CTS into the λΠ-calculus modulo theory makes these pieces of information explicit.

Changing the definition of CTS We started from a definition of CTS already studied in the litterature [Bar99a, Las12, Ass15b] but our results suggest that the current definition could be changed. Our reason to be conservative over this definition was to have the opportunity to study interoperability between CTS not only from one specification to another but also from one type system to another. But we see that being conservative requires to introduce some technicality. We suggest that the CTS definition could be changed, in particular we propose here two modifications:

- By restraining the specification to be in normal form (Definition 4.2.1)
- By adding the following typing rule:

$$\frac{\Gamma \vdash_{\mathscr{C}} \mathbf{wf} \qquad s \in \mathcal{S}_{\mathscr{C}}^{\top}}{\Gamma \vdash_{\mathscr{C}} s : s_{\infty}}$$

  where s_∞ is a specific sort not in $\mathcal{S}_{\mathscr{C}}$.

We have shown in Chapter 2 several theorems which explain that by doing so, we are defining a type system quite similar to the classic definition as defined in Chapter 1. The chapters 4 and 6 suggest that all CTS should be in normal form. Notice that all the specifications we have described which are behind the type system of concrete proof systems are already in normal form. This kind of incongruity is already present in the definition of PTS. Afterall, non-functional PTS are only functional CTS. This is reflected in the litterature where many papers only considered functional PTS.

The definition of semantic CTS (Definition 3.3.1) and our embedding to the λΠ-calculus modulo theory(Definition 6.1.2) suggests that this special sort s_∞ already exists somehow and is just implicit in the current definition of CTS. Moreover, this would recover a symmetry between top-sorts with respect to the subtyping rules: Only one subtyping rule would be needed.

Also, this notion of weak equivalence (Definition 2.1.12) defined in Chapter 2 is needed because with the current definition, typing context cannot contain variables typed by a top-sort.

Hence, is the current definition of CTS too general? If so...

What should be the "true" definition of CTS?

Part II

Interoperability in Dedukti: A case-study with Matita's arithmetic library

Chapter 8

Dedukti: An implementation of λΠ-calculus modulo theory

In Chapter 5, we have defined PTS modulo which enhance PTS with a custom conversion generated from arbitrary equations on terms. We have defined λΠ-calculus modulo theory as the extension of LF for PTS modulo. From Cousineau & Dowek [CD07], we know that every PTS (and by definition all the PTS modulo) can be embedded into the λΠ-calculus modulo theory. The main result of Chapter 6 also proves that every CTS can be embedded into λΠ-calculus modulo theory in a sound way. However, the type checking of λΠ-calculus modulo theory is undecidable [Bla01] because of the conversion.

One way to recover decidability of type checking is to orient the equations as a convergent term rewrite system [BN99]. This is the idea behind Dedukti which is an implementation of λΠ-calculus modulo theory where equations are implemented as a convergent rewrite system. However, having rewrite rules instead of equations raises two issues:

- What are the limits of the rewrite system to keep the type checking decidable
- How to ensure that subject reduction is still valid

For the first problem, there are three limits:

- The *matching problem* needs to be decidable
- Checking that a rewrite rule is well-typed needs to be decidable
- The term rewrite system is terminating

For the second problem, this comes back to proving the injectivity of products. This can be solved if we manage to prove the confluence of the term rewrite system [ABC+16].

Ronan Saillard shows in [Sai15] that it is possible to have an implementation of λΠ-calculus modulo theory with Higher-Order rewrite rules (restricted to the pattern fragment as defined by Dale Miller [Uni91]) so that the matching problem is decidable but also the type checking of rewrite rules. However, it is left to the user to check whether the rewrite system is convergent, ensure the injectivity of products, and, if the rewrite system is meant to capture an equational theory, to check that it does so. This implementation is called Dedukti.

We will see that having an implementation that allows non-convergent rewrite systems is convenient as it is shown in Chapter 9 to define *meta rewrite systems*.

It has been shown that many encodings in λΠ-CALCULUS MODULO THEORY can be implemented also as a term rewriting system: Matita [Ass15b], HOL-Light [Ass15b] or Focalize [Cau16a]. These results showed that DEDUKTI can be effectively used as an independent type-checker, for instance to validate proofs generated by these systems.

8.1 DEDUKTI

The system DEDUKTI implements the algorithms described in Ronan Saillard's Thesis [Sai15]. DEDUKTI enhances LF with rewrite rules and as such also implements the λΠ-CALCULUS MODULO THEORY calculus as defined in Chapter 5. To make the type checking of a theory encoded in λΠ-CALCULUS MODULO THEORY decidable in DEDUKTI, the term rewriting system should be convergent. However, it is undecidable in practice to check these conditions and the design of DEDUKTI does not enforce these conditions to be true. In particular, this means that the specification of DEDUKTI when the term rewriting system is not confluent or terminating is not properly well-defined (subject-reduction may not hold anymore). In practice, the system can loop, or return an error while the term is actually well-typed (if the system is not confluent for example). To overcome this issue, it is possible using DEDUKTI to call an external termination or confluence checker. The version 2.7 of DEDUKTI is compatible with the `TPDB` format [MNS19] which is used by many confluence checkers such as CSI^{HO} [NFM17] or `ACPH` [ACP16]. For termination, there exists currently only one external termination checker compatible with DEDUKTI: `SCT` [BGH19]. In practice, these tools work well for simple encodings in DEDUKTI. But for concrete encodings such as the one of CTS, it is unlikely that a termination or confluence checker could be used to check these properties automatically. A reason for that is that if the encoding of a logic is conservative (Definition 5.3.2), then the confluence and termination of the term rewriting system induced by the encoding implies the consistency of that logic. In particular, automatically checking the termination of the term rewriting system encoding CALCULUS OF CONSTRUCTIONS would imply the normalization of that same calculus.

The freedom offered by DEDUKTI by checking neither confluence nor termination is actually a real benefit since it allows experimentations really easily. Moreover, the tool DKMETA which will be presented in Chapter 9 uses this liberty to its own advantage for defining a meta rewrite system. However, one should be careful because subject reduction could be lost easily.

In practice, when a logic is encoded in DEDUKTI using a specific term rewrite system, checking that the system is convergent and does not break the injectivity of products is not enough to guarantee that DEDUKTI can check the proofs coming from that logic. There are two reasons for that:

- First, the mechanism of definition in DEDUKTI is implemented using rewrite rules. Hence, a proof should be a total function. This check is not done (yet?) in DEDUKTI because it requires making a distinction between symbols used to define a logic and symbols which are actually theorems inside this logic.

- Secondly, our shallow embedding does not encode (yet?) some requirements related to inductive types for example as the positivity criterion for inductive types (see Section 8.4.2).

The next section is devoted to the current implementation of DEDUKTI (version 2.7). We do not prove any meta-theory property in this section and refer the reader to Saillard's Thesis [Sai15].

8.1.1 Syntax of Dedukti

We briefly present the concrete syntax of Dedukti that we will use in the remaining part of this manuscript. One may find the full syntax at https://github.com/Deducteam/dedukti/blob/master/syntax.bnf.

To declare a type Nat of sort $\star$ we write

```
Nat : Type.
```

where `Type` is a keyword for $\star$. We can also add constant symbols such as 0 and S:

```
0 : Nat.
S : Nat -> Nat.
```

We use `->` to denote products as in $A \to B$. The same arrow is used for dependent products. By default these symbols are *static* (in opposition to *definable* symbols). In Dedukti, it is not allowed to add a rewrite rule on static symbols (the static symbol appears at the the head of the pattern). This way, a static symbol is automatically injective and this information can be used by Dedukti's type checker for rewrite rules. To declare a definable symbol, we need to use the keyword `def` as in:

```
def plus : Nat -> Nat -> Nat.
```

Such a function can be defined with rewrite rules as shown below:

```
[x] plus 0 x --> x.
[x,y] plus (S x) y --> S (plus x y).
```

We use `-->` for rewrite rules as in $\hookrightarrow_\beta$. In square brackets, we put the local variables of the rewrite rules (the ones in the local typing context Δ in $\lambda\Pi$-calculus modulo theory). In Dedukti, a definition can be given via a rewrite rule as

```
def 1 : Nat.
[] 1 --> S 0.
```

But Dedukti uses a syntactic sugar for this:

```
def 1 : Nat := S 0.
```

The very same mechanism is used to prove a theorem:

```
def thm : type := proof.
```

Finally, we have lambdas and applications:

```
A : Type.
g : A -> A.
def f : A -> A := x : A => g x.
```

Notice this time we use the arrow `=>` for the λ-abstraction.

Higher-order rewrite rules

The left-hand side of a rewrite rule in Dedukti is a *pattern* in the sense of Miller et al [Uni91]. A pattern is a subset of terms for which the matching problem is decidable. In Dedukti, patterns can also be nonlinear. An informal description of a patterm is given below.

Definition 8.1.1 (Pattern in Dedukti)
A pattern in Dedukti is roughly defined by the following principles.

1. *Always start by a definable symbol.*
2. *May contain local variables (in square brackets).*
3. *A local variable may have several occurences on the left-hand side.*
4. *A local variable can be applied to bound variables (bounded by a λ-abstraction in a pattern). Such local variable is said High-Order local variable.*
5. *Each higher-order local variable can be applied only to bound variables pairwise distinct*

Moreover, patterns may contain brackets. We postpone the semantics of brackets and nonlinear rewrite rule below.

Remark 26 *Dedukti allows wildcards in a pattern. This is just syntactic sugar for a local variable which is not used on the right-hand side of a rewrite rule.*

A toy example using patterns which higher-order logcal variable is the derivative of functions. For example:

```
1  Nat : Type.
2  0 : Nat.
3  S : Nat -> Nat.
4
5  def 1 : Nat := S 0.
6  def 2 : Nat := S 1.
7
8  def plus : Nat -> Nat -> Nat.
9  [x] plus 0 x       --> x.
10 [x,y] plus (S x) y --> S (plus x y).
11
12 def derivative : (Nat -> Nat) -> Nat -> Nat.
13 []    derivative (x => x)          --> x : Nat => 1
14 [F]   derivative (x => F)          --> x : Nat => 0
15 [F,G] derivative (x => plus (F x) (G x)) -->
16       x => plus (derivative (z : Nat => F z) x) (derivative (z : Nat => F z)
         ↪  x).
17
18 #CHECK (derivative (x : Nat => plus x x)) == (x : Nat => 2).
```

In this example, we define a function `derivative` which aims to compute the derivative of functions of type `Nat -> Nat`. For example the normal form computed by thos rules for the function `derivative (x => plus x x)` is `x => S (S 0)` which is what we expected. Notice that if a higher-order local variable is not applied to a bound variable, then this bound variable

cannot appear in the substituted term for this variable. This means that in `derivative (x => F)`, the variable `F` does not depend on `x`. If one had written `derivative (x => F x)` this means that the variable `x` may or may not appear in `F`. We will see more uses of this feature in Chapter 9.

Remark 27 *Some restrictions are implemented in* Dedukti *about this pattern fragment. One is that in* Dedukti*, a pattern always starts with a symbol. Hence, a λ-abstraction itself is not a* Dedukti *pattern but is a pattern defined by Miller in [Uni91]. Other restrictions are used to circumvent the fact that* Dedukti *is not modulo $\beta\eta$ which we will not detail here [Sai15].*

As we have seen in the example above with the command `#CHECK`, Dedukti defines a set of commands which are used mainly for debug usage. We will not detail these commands here and refer the reader to the official documentation instead: https://github.com/Deducteam/Dedukti.

Non-linear rewrite rules and brackets

Miller's patterns have been extended with two other features: *brackets* and *non-linear* rewrite rules. These two features enable a restricted version of conditional rewriting.

Brackets are an old feature which are now–for most of the use cases we know–outdated either by the algorithm to type check rewrite rules which is described below or by non-linear rewrite rules. A rewrite rule may contain an arbitrary term between brackets as a term which is not a Miller pattern as: `[F,G] s (x => F G)`. The semantics is the following: A bracket is replaced by a fresh variable `X` and a constraint `X = f (x => b)` is added. Every time Dedukti tries to apply a rewrite rule, then it checks whether the constraint introduced by the bracket is satisfied. If it does, it goes on, otherwise the type checking fails with an error message. The purpose of a bracket is to help the type checker to type check a rewrite rule, and is used as an assertion: Any term which matches this rewrite rule has to satisfy this constraint since this constraint has been used for the type checking.

Non-linear rewrite rules allow a variable to appear several times in a pattern. In that case, the semantics is the following: Each non-linear variable is replaced by a fresh variable with the constraint that the terms which match these variables are convertible. For example if one declares the rule `[x] f x x --> 0.`, then the term `f (2 + 2) 4` reduces to `0`.

8.1.2 Type checking and Subject reduction in Dedukti

One may notice that, since a rewrite rule is not symmetric, the type checking of a rewrite rule does not need to be symmetric anymore. This means that the rule $\mathscr{R}^{\mathbf{wf}}_{\equiv}$ does not check that the left-hand side of a rule **and** the right-hand side of the rules have the same type. Instead we only need to check that **whenever** the left-hand side is well-typed, the right-hand side is well-typed. This definition is particularly useful when we are faced with dependent types. The rule presented in Fig. 8.1 is well-typed in Dedukti, but it would not if we used the rule $\mathscr{R}^{\mathbf{wf}}_{\equiv}$. Indeed, the left-hand side is ill-typed because `n` and `m` are not equal. However, when the left-hand side is well-typed, we have `m` $=$ `n`, hence, the type of `v` is `vect m` which is equal to `vect n`. Hence, both sides of the rewrite rule have the same type. The type checking of a rewrite rule then becomes:

$$\frac{\Gamma, \Delta \vdash_{\mathscr{R}} B\sigma : T \text{ implies } \Gamma, \Delta \vdash_{\mathscr{R}} A\sigma : T \qquad \sigma \in \Delta \to \mathcal{T}}{\Gamma, A \hookrightarrow_{\Delta} B \vdash_{\mathscr{R}} \mathbf{wf}}\ \mathscr{R}^{\mathbf{wf}}_{\hookrightarrow}$$

```
nat : Type.

0 : nat.

S : nat -> nat.

vect : nat -> Type.

nil : vect 0.

A : Type.

cons : n:nat -> A -> vect n -> vect (S n).

def tail : n:nat -> vect (S n) -> vect n.

[n,m,a,v] tail n (cons m a v) --> v.
```

Figure 8.1: Type checking rewrite rule

DEDUKTI implements an heuristic to find such substitution [Sai15][1]. In particular this heuristic uses the fact that a symbol is static and so injective. Saillard proved the following statement for subject reduction:

Theorem 8.1.1 (Saillard) *If DEDUKTI satisfies the injectivity of product, then subject reduction holds.*

As argued at the beginning of this chapter, checking the injectivity of product requires in general to have confluence for which we can use an external prover. However, there is a catch here. Usual theorems for confluence rely on termination. Hence, before proving confluence, one should first prove termination. But usual techniques to prove termination require proving subject reduction. But proving subject reduction needs the injectivity of product which requires proving confluence etc...

There are two known possible solutions to solve this issue: First, proving confluence, termination and subject reduction at the same time or proving confluence first without assuming termination.

What is used in practice is the second solution, however it requires complex techniques because one cannot use Newman's lemma anymore [FJ19].

8.1.3 Rewrite strategy

Having a decidable type checking is not enough in practice to recheck theorems coming from other systems such as MATITA or COQ. The reason is that to check that two terms are convertible, checking the syntactic equality modulo α of their normal form requires too much time. Hence, one needs to implement a strategy which is fast in practice. Such strategy requires to compute the Weak-head Normal Form of a term [AGM92]. This is what is also done in other proof

[1] Actually, some efforts are made to weaken this rule when a substitution does not exist a priori.

```
1  A:Type.
2
3  a:A.
4  def b:A.
5  def f:A->A.
6
7  [] b --> a.
8  [] f b --> a.
9
10 #EVAL[WHNF] (f b). (; f b ;)
```

Figure 8.2: WHNF in Dedukti

assistants such as Coq or Matita. However, in Dedukti, since we have arbitrary user-defined rewrite rules the notion of Weak-Head Normal Form is a bit different from usual.

Definition 8.1.2 (Dedukti WHNF)
A term t is in WHNF if there exists a finite sequence $(t_i)_{i\mathbb{N}_m}$ such that:

- $t_0 = t$
- t_m *is in SNF*
- $t_i \hookrightarrow_{\beta\Gamma} t_{i+1}$ *such that the reduction does not appear at the head of* t_i

The reason for this definition is to take into account non confluent rewrite systems.

Example 8.1 *In the example of Fig. 8.2 we have a non confluent rewrite system. Without the first rewrite rule, the WHNF of* `f b` *would obviously be* `a`*. However, because of this first rewrite rule, the system is not confluent. The definition of WHNF says that* `f b`*,* `f a` *and* `a` *are all in WHNF. In Dedukti, the first is chosen though. If instead, we add a rule* `f a --> a` *then* `f b` *would not be a WHNF anymore.*

To compute the WHNF in an efficient way, Dedukti implements a rewrite engine that is closed from the one implemented in Matita and described in [ARCT09]. Some changes are done to take into account the other features of Dedukti: Higher-Order rewrite rules, non-linear rewrite rules and brackets. Currently, there is no documentation of the rewrite engine excepts the code itself.

Decision Trees

Another optimization implemented in Dedukti are decision trees. Decision trees were proposed by Luc Maranget [Mar08] to compile OCaml pattern matching. His algorithm has been implemented for Dedukti by Ronan Saillard and refined by Gabriel Hondet [Hon19]. A decision tree is a data-structure that implements a heuristic to choose a rule to use when the reduction engine encounters a definable symbol. Such symbol may have several rules and decision trees are better in general than trying to match the first rule, if it fails then try the second rules etc... The disadvantage of decision trees is that they tend to complexify the code of the rewrite engine and make it less easy to extend.

```
type red_cfg = {
  select   : (Rule.rule_name -> bool) option;
  nb_steps : int option; (* [Some 0] for no evaluation, [None] for no bound *)
  target   : red_target;
  strat    : red_strategy;
  beta     : bool;
  logger   : position -> Rule.rule_name -> term Lazy.t -> term Lazy.t -> unit
}
(** Configuration for reduction.
    [select] = [Some f] restrains rules according to the given filter on names.
    [select] = [None] is the default behaviour (all rules allowed).
    [nb_steps] = [Some n] Allows only [n] reduction steps.
    [nb_steps] = [None] is the default behaviour.
    [target] is the normal form to compute.
    [strat] is the reduction strategy.
    [beta] flag enables/disables beta reductions.
    [logger] is the function to call upon applying a reduction rule.
*)
```

Figure 8.3: Description of the `red_cfg` type

8.1.4 Extensions to the rewrite engine

In our work for interoperability, we needed to have some control over the rewrite engine strategy of Dedukti. For example, we needed to control the number of steps the rewrite engine could do, which rules were allowed and sometimes also the convertibility test. For these reasons, we have made a few changes to the rewrite engine of Dedukti which we mention below:

A rule has a name: For Dkmeta (see Chapter 9) we wanted to control which rewrite rules could be used and the ones that are not. This was hard, because in Dkmeta, the rewrite rules that are used to type check a file, and which ones could not are different and Dedukti was not able to make such difference. Our solution to overcome this issue was to add a name to a rewrite rule. Then, the rewrite engine takes a predicate over these names to know whether a rule can be used or not.

Having names for rewrite rules has another interest for debugging. It is easier to debug a Dedukti program if we know which rules have been used. As it might be cumbersome to give a name to every rewrite rule, names are optional in the user syntax, and a default one is generated in that indicates the line and the file where the rewrite rule has been defined.

The rewrite strategy: Exporting proofs to OpenTheory (see Chapter 12) requires having a trace of the computation done by Dedukti. To compute this trace, we needed to control the strategy so that Dedukti computes step by step. This is actually a tricky task to implement because to compute the WHNF, we sometimes need to compute under the head of a term (to check whether a constraint due to brackets or non-linear variables is satisfied). Therefore, the rewrite engine has now two functions to compute the WHNF: One which is used in practice to type check a Dedukti term using the default strategy, and one which we use for our purpose where we can parameterize the rewrite engine with the strategy we want. In OCaml, the structure which parameterizes the rewrite engine is given below in Fig. 8.3:

Changing the convertibility test: To implement the algorithm presented in Chapter 2, we needed to instrument the convertibility test of Dedukti. However, this requires maintaining a fork of Dedukti which is not convenient at all in practice. Instead, we have decided that the type checker should be an OCaml functor over a rewrite engine. This will be handy in Chapter 10 where we present the tool Universo. Universo needs to patch the default convertibility test of Dedukti to compute the free CTS (Definition 2.3.3). The OCaml interface of the functorized rewrite engine of Dedukti is defined in Figure. 8.4.

This interface as a default implementation used by the default type checker of Dedukti. The rewrite engine interface is split into two interfaces: One to reduce a term, and one to check whether two terms are convertible. In practice, the implementation of one module depends on the other and we have used recursive modules to implement this interface. The benefit of this interface, is that we can change the convertibility test without having to re-implement a function to compute the WHNF or the SNF of a term.

Private symbols: Private symbols[2] have been introduced to simulate proof irrelevance in Dedukti without having to modify its rewrite engine. A simple idea to simulate proof irrelevance is to rewrite all the proofs to a canonical proof. Such canonical proof could be given by a symbol which gives a proof to any proposition. However, this symbol makes the logic inconsistent because it gives a proof of `False`. Private symbols is a feature which allows to declare a symbol private which limits its scope. In Dedukti, a private symbol can be used only in the module (a file) in which it is declared. For all the other modules, this symbol cannot be written directly. However, it may appear through reductions. Hence, one could make the symbol which gives a canonical proof as *private*. In this way, we can ensure that outside the logic, such symbol cannot be used to write a proof of false. To ensure that any proof rewrites to this canonical proof, one needs to change the encoding function to ensure that any proof starts with a specific symbol. This is shown in example Fig. 8.5: In this encoding, any irrelevant proof is assumed to be encoded with the symbol `make_proof`.

Private symbols are used in the new encoding for CTS in Dedukti presented in Section 8.3.

8.2 Embedding of STT∀ in Dedukti

We present here an embedding of STT∀ into Dedukti. We already proposed an embedding of STT∀ into λΠ-calculus modulo theory using the CTS embedding. Here, we present in Figure. 8.6 a simpler embedding for STT∀ in Dedukti.

Because we saw in Chapter 7 that STT∀ is also a CTS, we will see in Chapter 9 how we can go from this encoding to the CTS embedding in Dedukti and vice versa.

Lemma 8.2.1 *The rewrite system of STT∀ is terminating and confluent.*

Proof *To prove the termination, the system is right-linear and in every rule, one of the symbol* `arrow`, `forallK`, `forallP`, `impl` *or* `forall` *disappears. The system is left-linear and orthogonal, hence it is confluent.*

Theorem 8.2.2 *The rewrite system of STT∀ and β reduction is confluent.*

Proof *The confluence is a direct consequence of 8.2.1 because the system has no critical pair with β and is left-linear [vOvR94].*

[2] A new nomenclature tends to call these symbols *protected*.

```
type constr =
  | Linearity
  | Bracket

type convertibility_test = Signature.t -> term -> term -> bool

module type S = sig
  val are_convertible  : convertibility_test
  (** [are_convertible sg t1 t2] checks whether [t1] and [t2] are convertible
      or not in the signature [sg]. *)

  val constraint_convertibility : constr -> Rule.rule_name ->
  ↪  convertibility_test
  (** [constraint_convertibility cstr r sg [t1] [t2] checks wehther the [cstr]
  ↪  of the rule [r]
      is satisfiable. Because constraints are checked once a term has matched
↪  the pattern,
      satisfying a constraint comes back to check that two terms are
↪  convertible *)

  val conversion_step : Signature.t -> term * term -> (term * term) list ->
  ↪  (term * term) list
  (** [conversion_step sg (l,r) lst] returns a list [lst'] containing
      new convertibility obligations.
      Raise [NotConvertible] if the two terms are not convertible. *)

  val reduction : red_cfg -> Signature.t -> term -> term
  (** [reduction cfg sg te] reduces the term [te] following the configuration
  ↪  [cfg]
      and using the signature [sg]. *)

  val whnf : Signature.t -> term -> term
  (** [whnf sg t] returns the Weak Head Normal Form of [t]. *)

  val snf  : Signature.t -> term -> term
  (** [sng sg t] returns the Strong Normal Form of [t].
      This may loop whenever [t] is not strongly normalizing. *)
end
```

Figure 8.4: A simplified version of the rewrite engine interface for DEDUKTI

```
1 Prop : Type.
2 proof : Prop -> Type.
3 private hilbert : A : Prop -> proof A.
4 def make_proof  : A : Prop -> proof A -> proof A.
5 [A,prf] make_proof A prf --> hilbert A.
```

Figure 8.5: Example of proof irrelevance in DEDUKTI with private symbols

```
type    : Type.
def eta : type -> Type.

ptype    : Type.
def etap : ptype -> Type.
p        : type -> ptype.

[] eta --> t => etap (p t).

bool     : type.
def eps : eta bool -> Type.

arrow : type -> type -> type.
forallK : (type -> ptype) -> ptype.

[l,r] etap (p (arrow l r)) --> eta l -> eta r.
[f]   etap (forallK f)     --> x : type -> etap (f x).

forall  : t:type -> (eta t -> eta bool) -> eta bool.
impl    : eta bool -> eta bool -> eta bool.
forallP : (type -> eta bool) -> eta bool.

[t,f] eps (forall t f) --> x:eta t -> eps (f x).
[l,r] eps (impl l r)   --> eps l -> eps r.
[f]   eps (forallP f)  --> x:type -> eps (f x).
```

Figure 8.6: STT∀ in Dedukti

Conjecture 14 (STT∀ is terminating) *The rewrite system of STT∀ and β is terminating*

We think that this could be proved using results from [DHK01] which proves that the encoding of Simple Type Theory in Dedukti is terminating.

8.3 Embedding of CTS in Dedukti

The embedding of CTS defined in λΠ-calculus modulo theory has been implemented in Dedukti. This extends Ali Assaf's works [Ass15b] since we have a generic encoding of CTS into Dedukti while Ali Assaf defined only an encoding for Matita in Dedukti. Moreover, his encoding had also a conservativity issue (see Example 6.6) which is fixed in our encoding into λΠ-calculus modulo theory. We give here an implementation of the public and private signature as proposed in Chapter 6 in the syntax of Dedukti. The main point here is the implementation of the private signature of the λΠ-calculus modulo theory encoding as rewrite rules in Dedukti. We use here the notion of private symbols introduced in Section 8.1.4. The system is presented in Figure. 8.7.

However, we have neither proven the confluence nor the termination of this system yet. The reason is that we have few confluence results about term rewrite systems which are Higher-Order and also non-linear. In fact, there is a general result that any Higher-Order and non-linear

```
Sort     : Type.
Univ     : s : Sort -> Type.
def Term : s : Sort -> a : Univ s -> Type.

bool : Type.
eps  : bool -> Type.
true : bool.

def Axiom   :     Sort -> Sort -> bool.
def Rule    :     Sort -> Sort -> Sort -> bool.
def Cumul   :     Sort -> Sort -> bool.
def SubType : s : Sort -> s' : Sort -> Univ s -> Univ s' -> bool.

sinf : Sort.

def univ : s  : Sort -> s' : Sort -> eps (Axiom s s') -> Univ s'.
def prod : s1 : Sort -> s2 : Sort -> s3 : Sort -> eps (Rule s1 s2 s3) ->
a : Univ s1 -> b : (Term s1 a -> Univ s2) -> Univ s3.
def cast : s : Sort -> s' : Sort -> a : Univ s -> b  : Univ s' ->
eps (SubType s s' a b) -> Term s a -> Term s' b.

univ' (s : Sort) (s' : Sort) : Univ s'.
def prod' : s1 : Sort -> s2 : Sort -> s3 : Sort ->
a : Univ s1 -> b : (Term s1 a -> Univ s2) -> Univ s3.
def cast' : s : Sort -> s' : Sort -> a : Univ s -> b : Univ s' ->
Term s a -> Term s' b.

[s,s',p]      univ s s' p        --> univ' s s'.
[s1, s2,s3,p] prod s1 s2 s3 p    --> prod' s1 s2 s3.
[s1,s2,a,b,t] cast s1 s2 a b _ t --> cast' s1 s2 a b t.

[s]         Term _ (univ' s _)                --> Univ s.
[s1,s2,a,b] Term _ (prod' s1 s2 _ a b)        --> x : Term s1 a -> Term s2 (b x).
[s,a]       Term _ (cast' _ _ (univ' s _) _ a) --> Term s a.

def forall : s : Sort -> a : Univ s -> (Term s a -> bool) -> bool.
[] forall _ _ (x => true) --> true.
I : eps true.

[s1, s2]           SubType _  _  (univ' s1 _)     (univ' s2 _ )   --> Cumul s1 s2
[s1,s2,s2',a,b,b'] SubType _  _  (prod' s1 s2 _ a b) (prod' _ s2' _ a b') -->
forall s1 a (x => SubType s2 s2' (b x) (b' x)).
[a]                SubType _ _   a               a               --> true.

[A,t]             cast' _ _ A A t                  --> t.
[s, s', a, c, t]  cast' _ s' _ c (cast' s _ a _ t) --> cast' s s' a c t.
[s1,s2,A,B,a]  cast' _ s2 (cast' _ _ (univ' s1 _) _ A) B a --> cast' s1 s2 A B a.
[s1,s2,A,B,a]  cast' s1 _ A (cast' _ _ (univ' s2 _) _ B) a --> cast' s1 s2 A B a.
[s1,s2,s3,s4,a,b]
cast' _ _ (univ' _ _) (univ' s4 _) (prod' s1 s2 s3 a b) --> prod' s1 s2 s4 a b.
[s1,s2,s3, a, b]
prod' _ s2 s3 (cast' _ _ (univ' s1 _) (univ' _ _) a) b --> prod' s1 s2 s3 a b.
[s1, s2, s3, a, b]
prod' s1 _  s3 a (x => cast' _ _ (univ' s2 _) (univ' _ _) (b x)) -->
prod' s1 s2 s3 a (x => b x).
[s1,s2,s3,A,B,C,b]
cast' _ _ (prod' s1 s2 _ A B) (prod' s1 s3 _ A C) (x => b x) -->
x : Term s1 A => cast' s2 s3 (B x) (C x) (b x).
[s1,s2,s3,A,B,C,b,a]
cast' _ _ (prod' s1 s2 _ A B) (prod' s1 s3 _ A C) b a -->
cast' s2 s3 (B a) (C a) (b a).
```

Figure 8.7: DEDUKTI's encoding of CTS

system is not confluent [KdV89]. But we do not need to prove the confluence of the whole rewrite system, but only for terms which are in the image of the encoding.

Example 8.2 *Going back to example 6.2, first we need to provide the specification signature the* Calculus of Constructions *with three universes*[3]*:*

```
(; Sorts ;)

star     : cts.Sort.
box      : cts.Sort.
triangle : cts.Sort.

(; Axioms ;)

[] cts.Axiom star box     --> cts.true.
[] cts.Axiom box triangle --> cts.true.

(; Rules ;)

[] cts.Rule     star     star     star --> cts.true.
[] cts.Rule     star      box      box --> cts.true.
[] cts.Rule     star triangle triangle --> cts.true.
[] cts.Rule      box     star     star --> cts.true.
[] cts.Rule      box      box      box --> cts.true.
[] cts.Rule      box triangle triangle --> cts.true.
[] cts.Rule triangle     star     star --> cts.true.
[] cts.Rule triangle      box      box --> cts.true.
[] cts.Rule triangle triangle triangle --> cts.true.
```

Then we can apply the translation given in Chapter 6 on the judgment $\vdash_{\mathscr{C}^L} \lambda x : \mathbb{0}.\, x \Leftarrow \mathbb{0} \to \mathbb{1}$ *which gives the following and well-typed* Dedukti *term:*

```
def id :
  cts.Term
    triangle
    (cts.prod
       box
       triangle
       triangle
       cts.I
       (cts.univ star box cts.I)
       (__:(cts.Term box (cts.univ star box cts.I)) => cts.univ box triangle
       ↪  cts.I))
  :=
  cts.cast
    box
    triangle
    (cts.prod
       box
```

[3] We could to for an infinite number of universes but it is not necessary here.

```
17          box
18          box
19          cts.I
20          (cts.univ star box cts.I)
21          (__:(cts.Term box (cts.univ star box cts.I)) => cts.univ star box
            ↪  cts.I))
22       (cts.prod
23          box
24          triangle
25          triangle
26          cts.I
27          (cts.univ star box cts.I)
28          (__:(cts.Term box (cts.univ star box cts.I)) => cts.univ box triangle
            ↪  cts.I))
29       cts.I
30       (x:(cts.Term box (cts.univ star box cts.I)) => x).
```

This term is rather huge but we will see in Chapter 9 (Example 9.1) how it can be made shorter using meta-rewriting. Actually the normal form of this term is already pretty short:

```
1  def id :
2    (cts.Univ star) -> cts.Univ box
3    :=
4    x:(cts.Univ star) => cts.cast' box triangle (cts.univ' star box) (cts.univ'
     ↪  box triangle) x.
```

8.4 Embedding inductive types in Dedukti

The effective translation from arithmetic proofs written in Matita to STT∀ which is presented in Chapter 11 takes into account how inductive types from Matita are encoded in Dedukti. In this section, we give a high-level description of inductive types on a classical example which are natural numbers and how they can be embedded into the λΠ-calculus modulo theory. For a detailed presentation of inductive types we refer to [Har16] [MLS84].

One motivation for the introduction of inductive types in the Calculus of Inductive Constructions is that the induction principle for the usual encoding of natural numbers (Church's encoding) in the Calculus of Constructions is not derivable. This can be proved by looking at the normal form of its proof [Str92]. Inductive types provide a new construction to type theory to derive an induction principle for natural numbers but also for many other data structures such as the Boolean, the polymorphic lists etc...

Another direction which has been followed by the type theory behind Cedille [FS18] is to enrich the Calculus of Constructions with more primitive constructions which allows to derive the induction principle for a large class of inductive types.

8.4.1 Inductive types

The formalization of inductive types as it is done in [PM96] is very complex with plenty of details. Their full formalization is not necessary to understand their embedding into Dedukti and especially to understand the problems they will raise during the translation in STT∀ presented in Chapter 11. This is why we limit ourselves to a high-level description of inductive types using

simple examples on natural numbers, together with their encoding in λΠ-calculus modulo theory. The generalization of this embedding can be found in [BB12].

The Calculus of Inductive Constructions enriches the Calculus of Constructions with three constructions:

- The so-called inductive types
- A *match* operator to deconstruct an inductive type
- A *fixpoint* operator to enable generic recursion

To make the Calculus of Inductive Constructions a sound calculus, we restrain inductive types with a guard condition (see [PM96]) and a syntactic criterion is added to check the termination of a fixpoint (see [PM96]).

In a concrete system such as Coq, the recursive function `plus` can be defined this way:

```
Inductive nat : Type :=
| O : nat
| S : nat -> nat.

Definition plus : nat -> nat -> nat :=
fix add n {struct n} :=
match n with
| O => fun m => m
| S n => fun m => S (add n m)
end.
```

We do not present the typing rules associated to these constructions which are quite obvious on this example or can be found in [PM96]. However, we want to insist on the computational behavior of these new constructions which makes the terms `plus (S (S O)) (S (S O))` and `S (S (S (S O)))` convertible. This conversion generated by inductive types and recursive functions is called ι.

For the fixpoint in Calculus of Inductive Constructions, since the calculus needs to be sound, it needs to ensure strong normalization. This prevents the addition of the usual rule on fixpoints which is non terminating. To overcome this issue, a restriction is added that the recursive argument of f should start with a constructor. In this particular case, the two rules needed are:

$$\texttt{fix add O body} \hookrightarrow_\iota \texttt{body}\,\{\texttt{add} \leftarrow \texttt{fix add}\}$$

and

$$\texttt{fix add (S n) body} \hookrightarrow_\iota \texttt{body}\,\{\texttt{add} \leftarrow \texttt{fix add}\}$$

Hence the definition of a fix point cannot be unfolded if the recursive argument is a variable for example.

Finally, for the *match* construction, similar rules are added and behave as expected:

$$\texttt{match O with | O => f | S n => g n} \hookrightarrow_\iota \texttt{f}$$

and

$$\texttt{match (S n) with | O => f | S n => g n} \hookrightarrow_\iota \texttt{g n}$$

To sum up, the declaration of a new inductive type with n constructors enriches the ι conversion with:

- n rules to unfold a fixpoint
- n rules to unfold a match

8.4.2 Inductive types in Dedukti

In this work, we follow the translation presented by Ali Assaf in [Ass15b] which we also present here since we will mention this encoding in Chapter 11. We use again our example of natural numbers presented in the previous section to explain the embedding of inductive types in Dedukti. We also use Calculus of Constructions as our ambient logic in Dedukti using the CTS embedding.

The declaration `nat : Type` is translated in Dedukti as it would be done for a declaration in the Calculus of Constructions:

```
nat : cts.Term cts.sinf (cts.univ cts.box cts.sinf cts.I).
```

Then we add two declarations for the constructors:

```
0 : cts.Term cts.box nat.
S : cts.Term cts.box (cts.prod cts.box cts.box cts.box cts.I nat (_ => nat)).
```

Ali Assaf's encoding does not use a generic symbol `match` but instead adds a symbol `match` for every inductive type. This is not a real restriction because every time we use a `match` we know on which inductive type it is used. The type for `match` over natural numbers can be expressed in Dedukti as follows:

```
def  match :
  s      : cts.Sort                                          ->
  P      : (cts.Term cts.box nat -> cts.Univ s)              ->
  case_0 : cts.Term s (P 0)                                  ->
  case_S : (n:(cts.Term cts.box nat) -> cts.Term s (P (S n))) ->
  z      : cts.Term cts.box nat                              ->
           cts.Term s (P z).
```

Two remarks:

- The type is universe polymorphic meaning that it quantifies over a sort. Hence this encoding is outside the encoding of CTS as presented in Chapter 6.
- Since s is a sort variable, one cannot encode the products using the `cts.prod` constructor. Indeed, the product $(\star, s, s)$ is not a valid product in this encoding.

Expressing a type of Calculus of Constructions this way in Dedukti may lead to mistaken types. For example, applying the `match` symbol to the sort `cts.box` gives the following type in Dedukti

```
P      : (cts.Term cts.box nat -> cts.Univ cts.box)              ->
case_0 : cts.Term cts.box (P 0)                                  ->
case_S : (n:(cts.Term cts.box nat) -> cts.Term cts.box (P (S n))) ->
z      : cts.Term cts.box nat                                    ->
         cts.Term cts.box (P z).
```

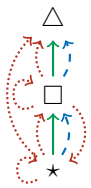


Figure 8.8: Extended version of Calculus of Constructions that supports strong induction

While this type is perfectly valid in Dedukti, it is not a valid type in the Calculus of Constructions! Indeed, the type of `P` requires that the sort `cts.box` has itself a type with a product towards this type. The fact that `match` can be applied to `cts.box` is actually known as the *strong elimination* and has been studied by Benjamin Werner in [Wer94] where Calculus of Constructions is enhanced with one universe and the corresponding products as pictured in 8.8.

Remark 28 *To avoid this universe polymorphic sort, it could be possible to duplicate the symbol* `match`*, one for every universe it is applied to. However, this may be problematic for interoperability. This remark will be detailed in Section 11.3.*

To each `match` symbol we associate a rule for each constructor. For the type `nat` the rules are the following ones:

```
[ s, P, case_O, case_S] match_nat s P case_O case_S O --> case_O.

[ s, P, case_O, case_S, n] match_nat s P case_O case_S (S n) --> case_S n.
```

Finally, the fixpoint operator is not translated as a fixpoint operator in Dedukti. Fixpoint operators as in Coq have a restriction that the reduction should be triggered only if the recursive argument starts with a constructor. Such restriction is not easy to encode in Dedukti. Hence, a fixpoint operator is always translated as a top-level recursive function. This is not completely satisfactory because two anonymous fixpoints could be convertible in Matita but are not in Dedukti since we gave a proper and different name to these fixpoints. But also, since there are translated as top-level functions, this requires closing the fixpoint by the current local typing context of Matita (λ-lifting) which again may break the conversion. In practice, these problems do not arise for the arithmetic library of Matita.

The idea implemented to translate Matita into Dedukti is to associate to each inductive type a filtering function which will trigger a computation only if the argument starts by a constructor. Hence, to each inductive type, a filtering function is added. Its type is really similar to the one of the `match` symbol:

```
def filter_nat :
  s           : cts.Sort                                    ->
  P           : (z:(cts.Term cts.box nat) -> cts.Univ s)    ->
  return      : (z:(cts.Term cts.box nat) -> cts.Term s (P z)) ->
  z           : (cts.Term cts.box nat)                      ->
                cts.Term s (P z).
```

which makes this constant also universe polymorphic. The third argument is the body of the function itself.

Then the `plus` function is translated as follows:

```
def plus :
      cts.Term cts.box
        (cts.prod cts.box cts.box cts.box cts.I
           nat
           (__ : cts.Term cts.box nat =>
            cts.prod cts.box cts.box cts.box cts.I
              nat
              (__1 : cts.Term cts.box nat =>
               nat))).

def plus_body :
      cts.Term cts.box
        (cts.prod cts.box cts.box cts.box cts.I
           nat
           (__ : cts.Term cts.box nat =>
            cts.prod cts.box cts.box cts.box cts.I
              nat
              (__1 : cts.Term cts.box nat =>
               nat))).

[n]
  plus n
  -->
  filter_nat box
    (x:(cts.Term cts.box nat) =>
     cts.prod cts.box cts.box cts.box cts.I nat
         (_:(cts.Term cts.box nat) => nat))
    plus_body
    n

[n] plus_body n -->
  m : cts.Term (cts.type cts.z) nat =>
  match_nat (cts.type cts.z)
    (__1 : cts.Term (cts.type cts.z) nat =>
     nat)
    m
    (p : cts.Term (cts.type cts.z) nat =>
     S (plus p m))
    n.
```

The `filter` function will delay the computation of `plus` until a value is given. Since we want to avoid a non-terminating system, the symbol `plus` cannot appear on the right-hand side of the rule and applied to a variable. The trick is to use an intermediate function `plus_body` which will be called after the `filter` function. The corresponding rewrite rule of the `filter` function will now trigger one step of computation. The rewrite rule that indeed performs the computation is given by the one on `plus_body` and this later calls recursively.

In this way, we can ensure that the rewrite rule on `plus_body` can be applied only if its argument starts by a constructor.

Finally, some remarks concerning this encoding proposed by Ali Assaf:

- For each inductive type, we need to introduce a `match` symbol specific to this inductive type. The reason is that the arguments given to a `match` symbol depend on the definition of the inductive type.
- The `match` symbol introduces a little bit of universe polymorphism. This is expected since if we gave a type to `match` in Calculus of Inductive Constructions, it would be itself universe polymorphic.
- The encoding does not have a general fixpoint as in Calculus of Inductive Constructions. Instead, there is a generic process to encode each recursive function and therefore the translation does not preserve fully the conversion.
- To avoid non-termination, this generic process introduces a `filter` function to ensure that any recursive call is done only if the argument starts with a constructor. This has also the disadvantage that for any recursive function, we need to have two symbols in Dedukti: to separate the recursive call and its implementation.

8.5 Future Work

Levels to solve the confluence/termination/subject reduction issue: The meta-theory of Dedukti has an issue related to the triptych: confluence/termination/subject reduction. A priori there is a circularity which is usually broken by proving first confluence without assuming termination. Well-structured derivation trees introduced in Chapter 3 give a new induction principle for CTS using levels. We think levels could be also used in PTS modulo and especially in Dedukti. In Dedukti, we think that we could add a requirement to the well-structured definition which is that for any well-typed rewrite rule $l \hookrightarrow_\Delta r$ then for any σ such that $\Gamma, \Delta \vdash_{\mathscr{D}} l\sigma : A$ is derivable at level n, then $\Gamma, \Delta \vdash_{\mathscr{D}} r\sigma : A$ is also derivable at level n. This condition is similar to the one we have made for β reduction (see Definition 3.1.2). We think this idea could break the circularity mentioned previously. Indeed, now we can prove subject reduction by induction on the level first. Proving subject reduction at level $n+1$ requires to have the injectivity of product at level n, hence we need confluence at level n. Proving confluence at level n needs termination at level n. Finally proving termination at level n needs subject reduction at level n. Thus the circularity is broken as we did for the equivalence between typed and untyped conversion in Theorem 3.3.8.

Instrumenting the conversion: While we observe that most of the time, type checking a proof is really fast in Dedukti, it happens for some proofs that Dedukti is way slower than the original type checker. One case of this is a lemma in the Matita's arithmetic library called `le_fact_10` which is a proof that $10! \leq 2^8 \times 5! \times 5!$. The proof of this theorem takes less than a second to be type checked in Matita and more than 10 minutes to be type checked in Dedukti. The reason is that the proof term elaborated by Matita is a proof of $10! \leq (2^8 + 0) \times 5! \times 5!$ which is not syntactically the same. In Dedukti, the rewrite engine will compute the weak normal form of the right-hand side of this expression which is the unary representation of the number 3686400. In Matita the rewrite engine first reduces $2^8 + 0)$ to 2^8 and then realizes that the two expressions are the same.

This example may seem extreme, but actually also happens in the Coq's standard library and probably it will happen for bigger libraries such as math-comp. This is a problem for several reasons:

- It makes time-consuming to type check a whole library in Dedukti if this happens many times,

- Such a theorem makes interoperability harder because for interoperability a theorem may need to be type checked many times,
- Once exported, such a theorem may take a long time to be type checked in the target system.

Changing the heuristic of the convertibility test of Dedukti will work once but may fail in other cases since the optimal strategy is not computable in an efficient way. One may consider that the strategy of the original is part of the logic (for example if the original system is weakly normalizing) and hence, as a logical framework, the λΠ-calculus modulo theory should provide a way to instrument this rewrite strategy. This would mean enhancing the calculus with an (optional) trace which would guide the conversion in the λΠ-calculus modulo theory. It is not clear what this trace would be because it needs to have good properties with substitution but also even if it is optional, the trace should not lead to an explosion of the size of the term.

The encoding of inductive types: The current encoding for inductive types is not very pleasant for the following reasons:

- Preservation of conversion is lost during the translation because the fixpoint operator is not translated as an anonymous operator but instead as a top-level operator with a name.
- The *match* operator is translated as a universe polymorphic constant which is outside the CTS encoding we have presented. Because of this, the `match` operator can be applied to a sort `s` in a way that the type of `match s` is outside the specification provided by the user.
- The guard criterion for inductive types and the fixpoint termination criterion are not translated.
- A filtering function is added for each inductive type to ensure that the rules are terminating. This function has no antecedent in the original theory and tends to obfuscate the proofs. They may also raise an issue for interoperability (see Chapter 11).

Calculus of Inductive Constructions introduces inductive types with a *match* and *fixpoint* operators, but this is not the only way to encode inductive types. One may use primitive eliminators *a la* System T, use the $\mathcal{W}$-types [MLS84] [MP00], or use the *gentle art of levitation* [CDMM10] or even the constructors introduced in Cedille [FS18]. However, we think that changing the way inductive types are encoded into Dedukti will not bring a better solution because, first, it requires to encode the *match* and *fixpoints* constructions to the new encoding, this encoding may not be suitable to be exported and, second, it makes the translation of the original logic harder to understand and to maintain. In particular, it is not clear if the encoding presented in the litterature still aplly since, if we take the Coq system for example, it is not clear whether the translations proposed in [Gim94] to go from *match* and *fixpoints* to primitive eliminators are still working since many criteria involving the guard conditions changed.

A first idea to fix the problem we have raised is to have a real anonymous *fixpoint* operator. This idea has been explored by Gaspard Férey for the translation of Coq into Dedukti and gives promising results[4]. While the termination criterion is not encoded, there is only one fixpoint operator which also takes into account extensions of inductive types with parameters and mutual inductive types. This encoding also removes the filtering function. The disadvantage is that all the complexity is hidden in Dedukti and if there is a type checking error involving inductive types, the error message is generally not intelligible even for a Dedukti expert.

[4] https://github.com/Deducteam/CoqInE

Chapter 9

Rewriting as a Programming Language

Going from one logic to another requires designing algorithms that can be applied to proofs. In practice, these algorithms need to be implemented in some programming language. Choosing a good programming language allows to write the transformations in a clean and efficient way. Moreover, since the application for these algorithms is interoperability, we aim to express our program in a modular way with respect to the logic. We realize that a programming language such as OCaml is not that good for writing these transformations:

- It features a first-order representation for binders which is not convenient in practice[1].
- These algorithms are easier to write if we can match directly on Dedukti terms instead of their abstract syntactic representation.
- Modularity with respect to the logic is not convenient with a language such as OCaml because a small change to the logic often implies a recompilation of the OCaml program[2] In general, we have realized that if a translation between proofs using something specific to some logic (as public symbols which only exist in the encoding of CTS for example), then it is better to have a parameterized translation. In particular the specific part to the logic is easier to be written outside of OCaml so that the translation is indeed parametric and also easier to use.

Cauderlier already proposed that the rewriting engine of Dedukti itself could be used as a programming language: First, to write a partial proof transformation which tries to transform a classical proof into a constructive one [Cau16b], and second, to define a tactic language for Dedukti [Cau18]. His notion of tactics relies on introducing definable symbols which stand for meta-variables (a hole for a proof term). His framework declares several tactics as a (partially) defined symbol in Dedukti. Then, a user can try to instantiate a meta-variable by a proof term by rewriting this variable to a tactic. The interaction comes from the user which asks for the normal form of a meta-veriable to Dedukti and copy/paste the result into the original file.

In this chapter, we follow this direction and see that rewriting as a programming language can be used in an efficient way also to write proof transformations for interoperability. Using rewriting in that purpose is what we call *meta rewriting* which gives rise to the tool we present in this chapter: Dkmeta. Dkmeta is a small tool built around Dedukti's kernel that offers a way to use Dedukti's rewrite engine to rewrite Dedukti's terms using Dedukti itself. We show that the rewriting of Dedukti is expressive enough to write many proof transformations.

[1] The OCaml bindlib [LR18] library tries to fill that gap however.
[2] We are not considering the OCaml byte code machine which still introduce a compilation step.

The Dkmeta tool also introduces a quote/unquote mechanism which provides an efficient way to have syntactic pattern matching without having to modify Dedukti's kernel. We believe that Dkmeta with a programming language such as OCaml provides a nice combination to write proof transformations in a concise and modular way which does not require too many efforts for maintainability.

This chapter is organized as follows: In Section 9.1, we present Dkmeta, its syntax, and how it can be used from the command line. In Section 9.2, we explain the quote/unquote mechanism that we introduce in Dkmeta. In Section 9.3, we present the applications of Dkmeta. In Section 9.4 we present the changes we have introduced to the kernel of Dedukti so that Dkmeta does not depend on the implementation of the kernel. In Section 9.5, we present related works which involve meta programming. Finally, we present in Section 9.6 possible extensions to this Dkmeta tool.

9.1 Dkmeta

Dkmeta is implemented in OCaml as a tool which uses Dedukti as a library. It provides a user-interface with the command line and also an OCaml package that can be used as an OCaml library by other programs. The input of Dkmeta is first, a Dkmeta program–as a set of Dedukti rewrite rules–and secondly a set of files on which this program is applied. The output of Dkmeta is a set of files where every term has been normalized with respect to the Dkmeta program. The normalization procedure uses a call-by-value strategy and computes the strong normal form if it exists. Currently there is no check if the Dkmeta program is terminating or confluent (and in general, the latter property is false). Optionally, one can check that the rewrite rules are well-typed in the sense of Dedukti. If no meta rules are provided, Dkmeta just normalizes the files given by the user to the rules declared in thos files.

When doing meta-rewriting it is interesting to match against an atomic construction of Dedukti such as a product or an application of Dedukti. But this is not possible because a product cannot appear inside a pattern (see Definition 8.1.1) or an application involving two local variables. Or even in some cases we want to have access to the type of a term. To circumvent these limitations, Dkmeta offers a quote/unquote mechanism. Hence the user can match against a *quoted* product for example. Currently, three quoting functions are provided by Dkmeta(see Section 9.2). In Section 9.6, we discuss how these quoting/unquoting functions could be provided directly by the user and not hard-coded in Dkmeta.

Example 9.1 *Going back to our example 6.2, we have shown that the CTS encoding of the judgment* $\vdash_{\mathscr{C}^L} \lambda x : \mathbb{0}. x \Leftarrow \mathbb{0} \to \mathbb{1}$ *was big. One could instead uses its normal form, but it gives a term outside the public signature for CTS encoding which is not convenient for interoperability. Dkmeta can be used as a trade-off to have readable terms without getting out the the public signature.*

Let us say that a user whishes to use the following shortcuts instead:

```
def U0 := cts.Univ star.

def U1 := cts.Univ box.

def u0 := cts.univ star box cts.I.

def u1 := cts.univ box triangle cts.I.

def pi00 := cts.prod box box box cts.I u0 (_ => u0).
```

```
10
11 def pi01 := cts.prod box triangle triangle cts.I u0 (_ => u1).
12
13 def castpi00pi01 := cts.cast box triangle pi00 pi01 cts.I.
```

To do so, it needs to give meta rewrite rules *which in this case are just the definitions above but reversed.*

```
1 (; meta rewrite rules ;)
2 [] cts.Univ star --> U0.
3 [] cts.Univ box --> U1.
4 [] cts.univ star box cts.I --> u0.
5 [] cts.univ box triangle cts.I --> u1.
6 [] cts.prod box box box cts.I u0 (_ => u0) --> pi00.
7 [] cts.prod box triangle triangle cts.I u0 (_ => u0) --> pi01.
8 [] cts.cast box triangle pi00 pi01 cts.I --> castpi00pi01.
```

Dkmeta will normalize the big term (fully written in Example 6.2) and produces the following short term instead.

```
1 def id : cts.Term triangle pi01 := castpi00pi01 (x : U0 => x).
```

The example above is a bit artificial, in particular because these definitions could be directly implemented by the tool procuding Dedukti code through the CTS encoding. But after one proof transformation, all these definitions disappear since we are interested only in proof terms using the CTS public signature. Hence, Dkmeta can be used to apply these definitions between proof transformations.

Notice that if the user had used the following definition

```
1 def tpi01 := U0 -> U1.
```

then inversing this definition does not give a valid rewrite rule in Dedukti:

```
1 [] U0 -> U1 --> tpi01.
```

Because a product cannot appear inside a pattern. In Dkmeta we solve this problem using a quote and unquote mechanism.

9.2 Quoting and unquoting

Quote and unquote functions were added in Dkmeta for two purposes: To allow the user to write a Dedukti product $((x : A) \to B)$ as a pattern and also to introduce syntactic pattern matching (matching is not computed modulo β). These are not the only usage for a quote and unquote mechanism and they can be used for other purposes. Dkmeta introduces a quoting mechanism as it is done in languages such as Lisp [Ste90] which in our case avoids modifying the kernel of Dedukti. In Fig. 9.1 we represent how Dkmeta works with a quoting mechanism. In this picture, $\mathcal{R}$ contains the meta rewrite rules. Some examples in Section 9.3 explained how the quoting mechanism can be used in practice. However, using a quoting mechanism may introduce a cost in computing time that is not always wanted. For this reason, we have declared three quoting functions, each one being more general than the previous one but also longer to compute. The three quoting functions that we have implemented are: `prod` which allows to rewrite a product and the Dedukti sort `Type`. The second encoding `lf` introduces syntactic

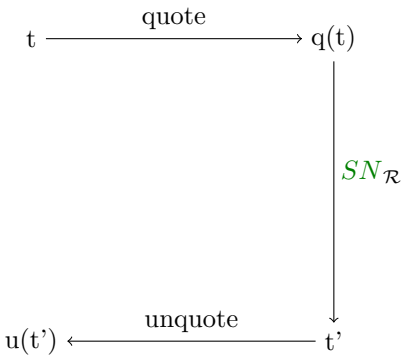


Figure 9.1: Dkmeta with a quoting and unquoting mechanism

pattern matching for every constructor of the language. Finally, the third encoding `ltyped` adds typing information on applications. The quoting functions of `prod` and `lf` do not require to have a type checker and hence, using this quoting function, Dkmeta is still fast to compute. However, for the last quoting function `ltyped`, we need to use the type checker of Dedukti. Hence it takes much more time for Dkmeta to compute the quotation of a term. We will use the following example (a version of Leibniz equality in STT$\forall$) to show the result of the quoting function.

```
1 def leibniz : A : type -> eta A -> eta A -> eta bool :=
2 A : type  =>
3 x : eta A =>
4 y : eta A =>
5 forall (arrow A bool) (P : (eta A -> eta bool) => impl (P x) (P y)).
```

For each of the quoting function below we give in details only the quote function on terms. The corresponding unquote function should be obvious. Moreover, we do not specify the quote function on patterns. One can check that the quote function is compatible with patterns (a pattern seen as a term is translated as pattern). In the definitions below, black symbols represent symbols introduced and specific to the quoting function.

9.2.1 Quotation for products

The purpose of this quote function is that the image of a product $(x : A) \to B$ by the quote function becomes a pattern. Therefore, a product can be rewritten using a meta rewrite rule. The quote function for product is named $[\![t]\!]^p$ (or `prod`) and defined as follows:

$$
\begin{aligned}
[\![\text{cst}]\!]^p &:= \text{cst} \\
[\![x]\!]^p &:= x \\
[\![\star]\!]^p &:= \mathbf{ty} \\
[\![f\ a]\!]^p &:= [\![f]\!]^p\ [\![a]\!]^p \\
[\![\lambda x : A.\, t]\!]^p &:= \lambda x : [\![A]\!]^p.\, [\![t]\!]^p \\
[\![(x : A) \to B]\!]^p &:= \mathbf{prod}\ [\![A]\!]^p\ (\lambda x.\, [\![B]\!]^p)
\end{aligned}
$$

Here is the result of this quote function applied to the definition of Leibniz equality.

```
def leibniz :
  prod.prod
    type
    (A:type => prod.prod (eta A) (__:(eta A) => prod.prod (eta A) (__:(eta A)
    ↪  => eta bool)))
  :=
  A:type =>
  x:(eta A) =>
  y:(eta A) =>
  forall (arrow A bool) (P:(prod.prod (eta A) (__:(eta A) => eta bool)) => impl
  ↪  (P x) (P y)).
```

A practical use of this quote function is presented in Section 9.3.2.

9.2.2 Quotation for syntactic pattern matching

This quote function aims to be used to get a syntactic pattern matching (to match against a Dedukti application for example). This pattern matching is of course not done modulo β. In this quote function, each constructor is prefixed with a symbol. This quote function is described below. Notice that the quoting for the product is similar to the one of the previous quote function.

$$
\begin{aligned}
[\![\mathrm{cst}]\!]^{\,l} &:= \mathbf{sym}\ \mathrm{cst} \\
[\![x]\!]^{\,l} &:= \mathbf{db}\ \mathrm{x} \\
[\![\star]\!]^{\,l} &:= \mathbf{ty} \\
[\![f\ a]\!]^{\,l} &:= \mathbf{app}\ [\![f]\!]^{\,l}\ [\![a]\!]^{\,l} \\
[\![\lambda x : A.\, t]\!]^{\,l} &:= \mathbf{lam}\ \lambda x : [\![A]\!]^{\,l}.\, [\![t]\!]^{\,l} \\
[\![(x : A) \to B]\!]^{\,l} &:= \mathbf{prod}\ [\![A]\!]^{\,l}\ \lambda x.\, [\![B]\!]^{\,l}
\end{aligned}
$$

The result of this quote function applied to the definition of Leibniz equality is displayed below:

```
def leibniz :
  lf.prod
    (lf.sym type)
    (A =>
     lf.prod
       (lf.app (lf.sym eta) (lf.var A))
       (__ => lf.prod (lf.app (lf.sym eta) (lf.var A)) (__ => lf.app (lf.sym
       ↪  eta) (lf.sym bool))))
  :=
  lf.lam
    (A:(lf.sym type) =>
     lf.lam
       (x:(lf.app (lf.sym eta) (lf.var A)) =>
```

```
13          lf.lam
14            (y:(lf.app (lf.sym eta) (lf.var A)) =>
15             lf.app
16               (lf.app (lf.sym forall) (lf.app (lf.app (lf.sym arrow) (lf.var A))
                 ↪  (lf.sym bool)))
17               (lf.lam
18                  (P:(lf.prod
19                        (lf.app (lf.sym eta) (lf.var A))
20                        (__ => lf.app (lf.sym eta) (lf.sym bool))) =>
21                   lf.app
22                     (lf.app (lf.sym impl) (lf.app (lf.var P) (lf.var x)))
23                     (lf.app (lf.var P) (lf.var y))))))).
```

Even if this quote function does not require any type checking, the fact that it produces a bigger term makes Dkmeta longer to normalize a term. A use of this quote function is presented in Section 9.3.4.

9.2.3 Quotation with a type annotation for applications

This quote function is almost the same as the previous quote function except that for an application $f\ a$, the quote function also adds the type of `f`. We do not use the same quotation function for the type inferred because it is not needed in practice. Indeed, we only need to encode the product and this is why we apply the quotation function for products on the type inferred. This quote function takes as input a term and a typed context and is presented below. For the λ-abstraction case and the application case, we have a side condition because the type inferred by Dedukti may not be a product. In practice we just compute the WHNF of the type which is always a product if the term is well-typed.

$$
\begin{aligned}
[\![\mathrm{cst}]\!]^{a}_{\Gamma} &:= \mathbf{sym}\ \mathrm{cst} \\
[\![x]\!]^{a}_{\Gamma} &:= \mathbf{db}\ \mathrm{x} \\
[\![\star]\!]^{a}_{\Gamma} &:= \mathbf{ty} \\
[\![f\ a]\!]^{a}_{\Gamma} &:= \mathbf{app}\ [\![(x:A) \to B]\!]^{p}\ [\![f]\!]^{a}_{\Gamma}\ [\![a]\!]^{a}_{\Gamma} && \text{where } \Gamma \vdash_{\mathscr{D}} f : (x:A) \to B \\
[\![\lambda x:A.\,t]\!]^{a}_{\Gamma} &:= \mathbf{lam}\ [\![(x:A) \to B]\!]^{p}\ \left(\lambda x:[\![A]\!]^{a}_{\Gamma}.\,[\![t]\!]^{a}_{\Gamma,x:A}\right) && \text{where } \Gamma \vdash_{\mathscr{D}} \lambda x:A.\,t : (x:A) \to B \\
[\![(x:A) \to B]\!]^{a}_{\Gamma} &:= \mathbf{prod}\ [\![A]\!]^{a}_{\Gamma}\ \lambda x.\,[\![B]\!]^{a}_{\Gamma,x:A}
\end{aligned}
$$

If we apply this quote function on our Leibniz example, we see that the size of the term explodes.

```
1  def leibniz :
2    ltyped.prod
3      (A:(ltyped.sym leibniz.type) =>
4       ltyped.prod
5         (__:(ltyped.app
6                (prod.prod leibniz.type (__:leibniz.type => prod.ty))
7                (ltyped.sym leibniz.eta)
8                (ltyped.var A)) =>
9          ltyped.prod
```

```
10          (__0:(ltyped.app
11                  (prod.prod leibniz.type (__0:leibniz.type => prod.ty))
12                  (ltyped.sym leibniz.eta)
13                  (ltyped.var A)) =>
14           ltyped.app
15             (prod.prod leibniz.type (__1:leibniz.type => prod.ty))
16             (ltyped.sym leibniz.eta)
17             (ltyped.sym leibniz.bool))))
18   :=
19   ltyped.lam
20     (prod.prod
21        (leibniz.eta A)
22        (x:(leibniz.eta A) =>
23         prod.prod (leibniz.eta A) (y:(leibniz.eta A) => leibniz.eta
          ↪  leibniz.bool)))
24     (A:(ltyped.sym leibniz.type) =>
25      ltyped.lam
26        (prod.prod (leibniz.eta A) (y:(leibniz.eta A) => leibniz.eta
        ↪  leibniz.bool))
27        (x:(ltyped.app
28              (prod.prod leibniz.type (__:leibniz.type => prod.ty))
29              (ltyped.sym leibniz.eta)
30              (ltyped.var A)) =>
31         ltyped.lam
32           (leibniz.eta leibniz.bool)
33           (y:(ltyped.app
34                 (prod.prod leibniz.type (__:leibniz.type => prod.ty))
35                 (ltyped.sym leibniz.eta)
36                 (ltyped.var A)) =>
37            ltyped.app
38              (prod.prod
39                 (prod.prod
40                    (leibniz.eta (leibniz.arrow A leibniz.bool))
41                    (__:(leibniz.eta (leibniz.arrow A leibniz.bool)) =>
                      ↪  leibniz.eta leibniz.bool))
42                 (__:(prod.prod
43                        (leibniz.eta (leibniz.arrow A leibniz.bool))
44                        (__:(leibniz.eta (leibniz.arrow A leibniz.bool)) =>
45                         leibniz.eta leibniz.bool)) =>
46                  leibniz.eta leibniz.bool))
47              (ltyped.app
48                 (prod.prod
49                    leibniz.type
50                    (A0:leibniz.type =>
51                     prod.prod
52                       (prod.prod
53                          (leibniz.eta A0)
54                          (__:(leibniz.eta A0) => leibniz.eta leibniz.bool))
55                       (__:(prod.prod
```

```
56                          (leibniz.eta A0)
57                          (__:(leibniz.eta A0) => leibniz.eta leibniz.bool))
                            ↪  =>
58                       leibniz.eta leibniz.bool)))
59                (ltyped.sym leibniz.forall)
60                (ltyped.app
61                   (prod.prod leibniz.type (__:leibniz.type => leibniz.type))
62                   (ltyped.app
63                      (prod.prod
64                         leibniz.type
65                         (__:leibniz.type =>
66                          prod.prod leibniz.type (__0:leibniz.type =>
                             ↪  leibniz.type)))
67                      (ltyped.sym leibniz.arrow)
68                      (ltyped.var A))
69                   (ltyped.sym leibniz.bool)))
70             (ltyped.lam
71                (leibniz.eta leibniz.bool)
72                (P:(ltyped.prod
73                      (__:(ltyped.app
74                             (prod.prod leibniz.type (__:leibniz.type =>
                               ↪  prod.ty))
75                             (ltyped.sym leibniz.eta)
76                             (ltyped.var A)) =>
77                       ltyped.app
78                         (prod.prod leibniz.type (__0:leibniz.type => prod.ty))
79                         (ltyped.sym leibniz.eta)
80                         (ltyped.sym leibniz.bool))) =>
81                    ltyped.app
82                      (prod.prod
83                         (leibniz.eta leibniz.bool)
84                         (__:(leibniz.eta leibniz.bool) => leibniz.eta
                            ↪  leibniz.bool))
85                      (ltyped.app
86                         (prod.prod
87                            (leibniz.eta leibniz.bool)
88                            (__:(leibniz.eta leibniz.bool) =>
89                             prod.prod
90                               (leibniz.eta leibniz.bool)
91                               (__0:(leibniz.eta leibniz.bool) => leibniz.eta
                                  ↪  leibniz.bool)))
92                         (ltyped.sym leibniz.impl)
93                         (ltyped.app
94                            (prod.prod
95                               (leibniz.eta A)
96                               (__:(leibniz.eta A) => leibniz.eta leibniz.bool))
97                            (ltyped.var P)
98                            (ltyped.var x)))
99                      (ltyped.app
```

```
100                    (prod.prod
101                       (leibniz.eta A)
102                       (__:(leibniz.eta A) => leibniz.eta leibniz.bool))
103                    (ltyped.var P)
104                    (ltyped.var y)))))))).
```

An application of this encoding is detailed in Section 11.5.

9.3 Applications of Dkmeta

We now explore in details how Dkmeta can be used in the context of interoperability. We review here three different applications. The next chapters detail more applications of Dkmeta.

9.3.1 STT∀ as a CTS and vice versa

The signature of STT∀ in Dedukti presented in Chapter 8 does not use the CTS signatature. However, we have shown in Chapter 7 that STT∀ could be seen as a CTS. Using the CTS specification of STT∀, one may translate the Dedukti STT∀ signature to the corresponding Dedukti CTS signature. This translation can be computed directly in Dkmeta as shown below.

```
1  [] sttfa.type --> cts.Term cts.triangle (cts.univ cts.box cts.triangle cts.I).
2
3  [] sttfa.ptype --> cts.Term cts.sinf (cts.univ cts.diamond cts.sinf cts.I).
4
5  [] sttfa.bool --> cts.univ cts.star cts.box cts.I.
6
7  [A] sttfa.p A --> cts.cast cts.triangle cts.sinf
8      (cts.univ cts.box cts.triangle cts.I) (cts.univ cts.diamond cts.sinf cts.I)
       ↪  cts.I A.
9
10 [A] sttfa.etap A --> cts.Term cts.diamond A.
11
12 [A] sttfa.eps A --> cts.Term cts.star A.
13
14 [A,B] sttfa.arrow A B --> cts.prod cts.box cts.box cts.box cts.I A (x => B).
15
16 [A,B] sttfa.impl A B --> cts.prod cts.star cts.star cts.star cts.I A (x => B).
17
18 [A,B] sttfa.forall A (x => B x) --> cts.prod cts.box cts.star cts.star cts.I A
   ↪  (x => B x).
19
20 [B] sttfa.forallP (x => B x) -->
21     cts.prod cts.triangle cts.star cts.star cts.I (cts.univ cts.box
       ↪  cts.triangle cts.I) (x => B x).
22
23 [B] sttfa.forallK (x => B x) -->
24     cts.prod cts.triangle cts.diamond cts.diamond cts.I (cts.univ cts.box
       ↪  cts.triangle cts.I) (x => B x).
```

To keep the translation simple we do not translate types operators but we could using the quote function for products. This is because the type of a type operator such as `list` is `sttfa.type -> sttfa.type` in Dedukti using the STT∀ signature.

The reverse translation from the CTS signature to the signature of STT∀ could also be written in Dkmeta. The idea is to take all the rewrite rules above and just reverse them. For example

```
1 [] sttfa.type --> cts.Term cts.triangle (cts.univ cts.box cts.triangle cts.I).
```

becomes

```
1 [] cts.Term cts.triangle (cts.univ cts.box cts.triangle cts.I) --> sttfa.type.
```

It is easy to see that the first translation defines a total function over the STT∀ signature in Dedukti. It is also true for the reverse translation but this is not obvious. This property relies on the property that in the CTS specification of STT∀ (Definition 7.2.3) the sort $\triangle$ has only one inhabitant which is $\square$.

9.3.2 Rewrite Products to compute canonical forms

Another application of Dkmeta is the computation of canonical forms with the quote function for products. Coming back to our `leibniz` example, the type of `leibniz` we have written is `A : type -> eta A -> eta A -> eta bool`. However, we could also have used the following alternatives:

- `A : type -> eta A -> eta (arrow A bool)`
- `A : type -> eta (arrow A (arrow A bool))`
- `etap (A => p (arrow A (arrow A bool)))`

which are all convertible in Dedukti. However, the last one seems better for at least two reasons:

- It does not contain any product, hence it is in the image of the translation as defined by the CTS translation in Definition 6.1.2 and the Dkmeta translation presented in Section 9.3.1. It is the same as saying that the term uses only symbols defined in the STT∀ signature.
- Looking at the head of the term, we know whether it is a type (it starts with `eta`) or a proposition (it starts with `eps`)

This is why we call the last representation the **canonical** representation. However, this representation is not always well-defined. For example in the encoding of a non-functional CTS, some types have several canonical representations: If $(s_1, s_2, s_3) \in \mathcal{R}$ and $(s_1, s_2, s_4) \in \mathcal{R}$ then the term $(x : \mathbf{T}_{s_1}\ A) \to \mathbf{T}_{s_2}\ B$ has two canonical representations: Either $\mathbf{T}_{s_3}\ (\boldsymbol{\pi}_{s_1,s_2,s_3}\ \mathbf{I}\ A\ (\lambda x.\, B))$ or $\mathbf{T}_{s_4}\ (\boldsymbol{\pi}_{s_1,s_2,s_4}\ \mathbf{I}\ A\ (\lambda x.\, B))$.

Computing the canonical representation of a term requires to invert the rules which define the symbols `eta` or `eps` in STT∀ and the symbol $\mathbf{T}$ in the CTS encoding. We can notice that all these rules that define these symbols are invertible in STT∀ and this is true also for functional CTS.

We give an example in STT∀ below. We recall below the rewrite rules which interpret the function symbols `arrow` and `forallK`.

```
[a,b] etap (p (arrow a b)) --> eta a -> eta b.
[f]   etap (forallK f) --> x: type -> etap (f x).
```

Their inverse could be something like

```
[a,b]  eta a -> eta b --> etap (p (arrow a b)).
[x,f] x:type -> eta (f x) --> etap (forallK f).
```

Since a product is not a pattern, we use the quoting function for products instead. When a quote function is used, the user needs to write meta rewrite rules on quoted terms. Because each DEDUKTI product is quoted using the symbol `prod.prod`, this gives the following rewrite rules.

```
[a,b] prod.prod (eta a) (x => eta b)     --> etap (p (arrow a b)).
[a,b] prod.prod (type)  (x => eta (f x)) --> etap (forallK (x => f x)).
```

Then, we can apply this technique on the Leibniz equality which produces as a result

```
def leibniz : etap (forallK (A => arrow A (arrow A bool))) :=
A : type  =>
x : eta A =>
y : eta A =>
forall (arrow A bool) (P : (eta (arrow A bool)) => impl (P x) (P y)).
```

In the case of the encoding of CTS, the inversion requires a function `rule` to compute a new sorts s_3 from s_1 and s_2. For example inverting the rewrite rule

```
[s1, s2, a, b] Term _ (prod s1 s2 _ a b) --> x : Term s1 a -> Term s2 (b x).
```

could be done as follow

```
[s1, s2, a, b] prod.prod (Term s1 a) (x => Term s2 (b x)) --> Term (rule s1 s2)
↪  (prod s1 s2 (rule s1 s2) I a (x => b x)).
```

It is also left to the user to define the function `rule`.

Hence DKMETA provides a concise way to compute the canonical form of a term. The meta rewrite system only needs to be defined once given the signature of some logic in DEDUKTI. In this example, we use the full expressivity of Miller's pattern fragment since the pattern of the last rule contains `x => Term s2 (b x)`. One could do the same thing in OCaml but it is tedious to write and of course the resulting program would be much longer.

9.3.3 Implicit arguments in DEDUKTI with DKMETA

Implicit arguments are arguments needed for type checking but that does not need to be provided by the user and could be inferred from the context. If we use again our definition of Leibniz equality in DEDUKTI, we could omit the first argument of the symbol `forall` (in this example we use the wild-card `_` as a placeholder for a meta-variable that should be instantiated):

```
def leibniz : A : type -> eta A -> eta A -> eta bool :=
A =>
x =>
y =>
forall _ (P => impl (P x) (P y)).
```

The type for the first three λ-abstractions can be easily inferred from the type of the `leibniz` constant. Moreover, Dedukti's kernel can also infer the type of the λ-abstraction for the variable `P` by type checking the right hand side. However, even if Dedukti had a representation for meta-variables, instantiating this meta-variable is hard. Let us denote this meta-variable as `?1`. The Dedukti's type checker processes the term `P => impl (P x) (P y)` to infer a type for the abstraction in the context where `impl:eta bool -> eta bool -> eta bool`, `x:eta A` and `y:eta A`. In particular, the type inferred for `P` is `eta nat -> eta bool`. Now, the unification problem that Dedukti needs to solve to instantiate the meta-variable `eta ?1` is: `eta ?1` $\stackrel{?}{=}$ `eta nat -> eta bool`. This problem is hard because Dedukti needs to guess that `?1` is `arrow nat bool`. This could be solved by computing the canonical representation of a type using the technique we presented above. Indeed, the canonical representation of `eta nat -> eta bool` is `eta (arrow nat bool)`. Now, the problem becomes solvable if we assume that `eta` is injective which is true in the STT$\forall$ encoding.

The main advantage of this technique is that it does not need to change Dedukti's kernel. Moreover, the unification algorithm that would be implemented is predictable and very simple in this case.

This example could be pushed a bit further. Dkmeta can be used also so that the user does not need to enter the wildcards for implicit parameters manually. Hence, to imitate the behavior of Coq for example, one could declare two symbols `@forall` and `forall`. The first one, has no implicit parameters, while the second one would have one implicit parameter. To make the connection between the two symbols, one could use the following meta rewrite rule

```
1 [P] forall P --> @forall _ P.
```

This usage of Dkmeta would be really similar to a preprocessor.

9.3.4 Compute Traces

Knowing that two terms are convertible, it might be interesting to have a trace which explains how these two terms are convertible. Such traces have several applications. This thesis explores two of them:

1. Going from Calculus of Inductive Constructions to STT$\forall$ requires to remove the rewrite rules that are introduced to encode recursors and fixpoints. Each application of a rewrite rule requires to compute the context where this rewrite rule has been used. This information can be recovered from the trace.

2. Exporting our proof from STT$\forall$ to OpenTheory requires to compute a trace for every δ and β rule that has been used.

But there exist other applications, in particular, for debugging. It is always possible to instrument the rewrite engine of Dedukti to compute such traces. However, this tends to slow down the rewrite engine even if this information is not needed. Moreover, instrumenting the rewrite engine to compute this context is not always an easy task since it requires a deep understanding of the rewrite engine. This instrumenation could be broken easily if the rewrite engine is modified (for example by introducing more sharing in terms). Since performance is not a primary criterion when a trace needs to be computed, we think that such functionality should be instrumented outside the kernel. In our case, we present an implementation with Dkmeta. This also has the advantage that we do not need to manipulate the context and the implementation details such as De Bruijn indices explicitly. For this application, the quote function for products

is not sufficient because we need to match against a syntactic application for example which is not possible in DEDUKTI since matching is done modulo β. This is why we use the `lf` encoding instead.

A *trace* explaining why two terms A and B are convertible can be defined informally as a list of *steps*. A *step* is composed of four pieces of information:

- The name of the rule,
- The syntactic context where it is applied,
- The substitution,
- The side (left or right).

The substitution could be recovered from the context and the name of the rule but it is in general preferable to have this information directly. The computation of a trace requires that DEDUKTI can compute only one rewrite step at a time. In this example, we will focus on recovering the context as a higher-order function. Getting the syntactic context to get a full trace is the most difficult part since all other information can be easily recovered using the context. In this example, we assume that we have two terms A and A' which are convertible up to one computation step modulo a rewriting system $\Gamma \cup \beta$. We want to compute their common context. To recover the context, we define two meta functions `get_context` and `get_context'`[3]. `get_context` is applied to two terms A and A' and call `get_context'` with the variable `h` which is the *hole*. `get_context'` is defined only for products, application and abstractions. We also use a non-linear rule to get rid of the common part easily.

```
[h,t,t'] get_context t t' --> lf.lam (h => get_context' h t t').

[h,t]
get_context' h t t
--> t.

[h,f,f']
get_context' h (lf.lam (x => f x)) (lf.lam (x => f' x))
--> lf.lam (x => get_context' h (f x) (f' x)).

[h,f,f',a,a']
get_context' h (lf.prod A (x => B x)) (lf.prod A' (x => B' x))
--> lf.prod (get_context' h A A') (get_context' h B B').

[h,f,f',a,a']
get_context' h (lf.app f a) (lf.app f' a')
--> lf.app (get_context' h f f') (get_context' h a a').
```

After normalization, as we assume that the two initial terms A and A' differ only by one reduction step, there remains only one instance of a function `get_context' h t t'`. This instantiation can be removed thanks to one rewrite rule:

```
[h,t,t'] get_context' h t t' --> h.
```

Hence, this computation is a two-stage process. This could be achieved only in one stage but it requires more rules if we want a confluent system.

[3] Since we have an untyped rewrite system, we could use only one function symbol

9.4 Implementation of Dkmeta

Initially, Dkmeta was implemented as a *fork* of Dedukti meaning that it could be considered as another implementation of Dedukti. Its only feature was to automatize the normalization of terms according to a set of rewrite rules. However, in terms of software engineering, this is not convenient at all because each modification done on Dedukti needed to be merged into the Dkmeta fork. Therefore, we were interested in having a tool around the kernel of Dedukti. To do so, we have made minor modifications to the kernel of Dedukti so that Dkmeta could be extracted as an external library for Dedukti.

9.4.1 Kernel modifications to Dedukti

We have chosen to parameterize the reduction engine with a new type called `type red_cfg`. This type is implemented as follows:

```
type red_cfg = {
  select   : (Rule.rule_name -> bool) option;
  target   : red_target;
  beta     : bool;
  (* ... *)
}
```

The first field allows the specification a set of rules that the reduction engine can use. Notice that the type `Rule.rule_name` is a way to identify a rule in a unique way. Often, the user does not want to write the name of the rules and Dedukti has to invent a new name. If all the rules are allowed (default case), by default this parameter is set to `None`. The second field needs to be set to `Snf` by Dkmeta, since Dkmeta always computes the strong normal form. Finally, the third parameter allows the user to deactivate β reductions. This configuration can be passed to the reduction engine via a function called `reduction` which is implemented in the module `Env` :

```
val unsafe_reduction : t -> ?red:(Reduction.red_cfg) -> term -> term
(** [unsafe_reduction env red te] reduces [te] according to the reduction
↪  configuration [red].
    It is unsafe in the sense that [te] is not type checked first. *)
```

Another modification which was introduced by Dkmeta to the kernel is a flag called `fail_on_symbol_not_found`. This flag deactivates the default behavior of Dedukti of raising an error when a symbol is not found in the signature. This is not mandatory, but it really eases the use of Dkmeta. This flag allows the signature to consider all the symbols which are not present in the signature as static symbols, hence without definition.

9.4.2 Dkmeta, a library for Dedukti

We review the essential parts in the implementation of Dkmeta. The library defines a type `cfg` which is defined as:

```
type cfg = {
  mutable meta_rules  : RNS.t option;
  (** Contains all the meta_rules. *)
  beta                : bool;
```

```
  (** If off, no beta reduction is allowed *)
  encoding            : (module Encoding) option;
  (** Set an encoding before normalization *)
  env                 : Env.t
  (** Current environement *)
}
```

The first parameter `meta_rules` contains the set of meta rewrite rules that will be used to modify the terms. The second parameter allows the deactivation of β reductions. The third parameter allows the specification of a quoting mechanism (as an OCaml first-class module), this is detailed in Section 9.2. The last parameter is a DEDUKTI environment, it is used to interact nicely with DEDUKTI kernel. A default configuration is defined as follows:

```
(** Initliaze a configuration with the following parameters:
    [meta_rules] = None
    [beta]       = true
    [encoding]   = None
    [env]         = empty_signature (in particular the name is the empty string) *)
val default_config : cfg
```

The library offers mainly two functions:

```
  val meta_of_rules: Rule.untyped_rule list -> cfg -> cfg
  (** [meta_of_rules rs cfg] adds the meta_rules [rs] in the configuration [cfg] *)

  val mk_term      : cfg -> ?env:Env.t -> Term.term -> Term.term
  (** [mk_term cfg ?env term] normalize a term according to the configuration [cfg] *)
```

The first function is called to instantiate a DKMETA configuration on all the meta rewrite rules given by the user. The second method is the method which does the actual process by calling the rewrite engine of DEDUKTI on the term to normalize. This method translates a DKMETA configuration into a DEDUKTI's rewrite engine configuration.

9.5 DKMETA vs other meta-languages

DKMETA can be considered as a meta language for DEDUKTI. Many projects around typing systems have also developed their own meta language. We review here some of these projects.

9.5.1 λPROLOG

λPROLOG [FGH+88] is a programming language which could be used as a meta-language. This is already done in the COQ-ELPI project [Tas19] which introduces λPROLOG as a meta-language for COQ. The main advantage of λPROLOG with respect to DKMETA is the backtracking mechanism which is built into λPROLOG. In DKMETA, backtracking could be simulated but this is not convenient to write and also error-prone[4]. On the other hand, the main advantage of DKMETA is to allow rewrite rules which are not well-typed and also non-linear rewrite rules which give more flexibility to the user. As of today, we did not find any use of backtracking in DEDUKTI. But

[4] Writing a non-terminating program is really easy in DKMETA, especially when one implements a backtracking algorithm.

this might change in the future (for example if one wants to implement a refiner[5] for Dedukti). In that case, it is not clear whether backtracking should be added into Dkmeta or if one should change the meta language to use λprolog instead (or a version of λprolog for Dedukti).

9.5.2 Beluga

Beluga [Pie10] extends LF with a meta language which looks like a first-order language where the terms are contextual LF objects. Contextual LF objects means that an LF term is represented with its context. Such meta language can be used to write meta functions over LF terms as it is done in Dedukti. However, the purpose of this meta layer is not to compute with meta functions but rather to prove that these meta functions are correct. This could be used to certify that the proof transformation process is correct. For example, in Beluga it is possible to prove that the transformation of a term (encoded in LF) represented using Higher-Order Abstract Syntax to an term (also encoded in LF) represented with De Bruijn indices is correct. However, this function is hard to write because the system needs to be convinced that the function is well-defined. Hence, it is harder to write meta functions since the typing system in this case is limiting. Extensions to facilitate the writing of meta functions have been proposed such as Cocon [PTA+19] which makes the meta layer more expressive.

9.5.3 Meta-Coq

Meta-Coq [ABC+18] (former Template-Coq) is another project around Coq which aims to be a first step to certify Coq in Coq. In Coq, inductive types allow the representation of the abstract representation of Coq terms. Hence the Meta-Coq project provides this inductive type as well as two functions which code and decode Coq terms towards/from this inductive type. Hence Meta-Coq implements also a quoting mechanism similar to Dkmeta. However their purpose as for Beluga is different since they are interested in the certification of a type checker for Coq which could be written in Meta-Coq as a meta function.

9.6 Future work

Dkmeta could be improved in many ways.

9.6.1 Define new quoting and unquoting functions with Dkmeta

We saw three quoting/unquoting functions in Dkmeta. However, these quoting functions are currently hard-coded in OCaml. Thus, adding an quoting function is painful and requires to recompile Dkmeta. We believe that there is a generic way to declare a new quoting function in Dkmeta. The idea is that there is a more general quoting function which could be informally defined as an *encoding* of Dedukti in Dedukti. We will not define the encoding function $[\![\Gamma]\!]^{g}_{\Gamma}$ here, but the target signature would be

```
type : Type.

def eta : type -> Type.

ty : type.
```

[5] A tool which translate user's syntax to kernel's syntax. A typical task for a refiner is the elaboration of implicit parameters.

```
6
7  [] eta ty --> type.
8
9  prod : A : type -> (eta A -> type) -> type.
10
11 var : A : type -> eta A -> eta A.
12
13 lam : A : type -> B : (eta A -> type) -> (a:eta A -> eta (B a)) -> eta (prod A
   ↪  B).
14
15 def app : A : type -> B : (eta A -> type) -> eta (prod A B) -> a:eta A -> eta
   ↪  (B a).
```

All the other quoting function can be defined as meta rewrite rules from this encoding. For example the quoting function for products can be defined with the following meta system.

```
1  [A] eta A --> A.
2
3  [x] var A x --> x.
4
5  [A,B,f] lam A B f --> f.
6
7  [A,B,f,a] app A B f a --> f a.
```

Hence $[\![t]\!]^{p}$ could be defined in a term of $[\![\Gamma]\!]^{g}_{\Gamma}$ and the meta rewrite system presented above.

Doing this naively could introduce an unnecessary cost at run time because $[\![t]\!]^{p}$ does not need Dedukti's type checker while $[\![\Gamma]\!]^{g}_{\Gamma}$ needs to call Dedukti's type checker. However, by looking at the meta rewrite system, Dkmeta knows statically if the encoding defined by the user needs a type checker or no. It is sufficient to look at whether one of the parameters which need to be inferred by Dedukti appears on the right-hand side. If it does, then the encoding needs the type checker, otherwise, Dkmeta can introduce a *fake* term instead of the real type since it will be thrown away.

9.6.2 Termination and confluence

Dkmeta offers almost no guarantee about the meta rewrite system provided by the user. Because there exist tools to check confluence [NFM17] and termination [BGH19], it could be interesting to use these tools to check whether the rewrite system provided by the user is confluent and terminating. This could help fixing obvious flaws in the user's rewrite systems.

9.6.3 Extending the language of Dkmeta

Currently, the normalization process can be used only on Dedukti terms. However, the Dedukti language also features top level commands to declare parameters, rewrite rules, definitions or commands. In the current version of Dkmeta, these objects are not first-class and the user cannot manipulate them. Hence, it would be interesting to enhance the language of Dkmeta to see these objects as first-class citizens. This could be a first-step to implement a refiner in Dkmeta for example where the instantiation and the use of meta-variables would be done directly in Dkmeta and not in OCaml.

Chapter 10

Universo

In Section 2.3, we have described an incomplete algorithm about interoperability between CTS. The problem that this algorithm solves is: Given a derivable judgment $\Gamma \vdash_{\mathscr{C}} t : A$ and a CTS $\mathscr{C}'$ to decide whether this judgment can be derivable in $\mathscr{C}'$ via a judgment embedding (Definition 2.1.4). In this chapter, we provide an implementation–called Universo–of this algorithm in Dedukti using the CTS encoding described in Chapter 8. This algorithm can be seen as a generalization of Coq's algorithm to check that the floating universes constraints are consistent [Typ05]. The algorithm we have implemented can be summed up as follows:

1. Elaborate the judgment to replace every sort by a fresh variable,
2. Generate the free CTS (as defined in Section 2.3) by invoking Dedukti as a type checker for CTS,
3. Find a sort-morphism from this free CTS to $\mathscr{C}'$ using an SMT solver,
4. If a solution has been found, replace the fresh sorts generated at step 1 by their image through the sort-morphism found at step 3.

Roughly, the free CTS is a specification associated to a derivation tree which makes this derivation tree type checkable whe every sort are replaced with a fresh variable.

Coq's algorithm implements a particular case where the CTS specification $\mathscr{C}$' is fixed and is the one of Coq, namely $\mathscr{C}^{C}_{s_\infty}$ (Definition 1.5.14). Their algorithm cannot be extended easily for any CTS mainly because it relies on algebraic universes [Typ05].

However, many technical details arise when going from an algorithm to a concrete implementation. One detail we think is important is: How the same tool can be used for proofs encoded in two different logics? Our solution relies on the notion of public and private signatures we have presented in Chapter 6. The idea, is that two proofs comming from two different proof systems (e.g. Matita and Coq) have to use for CTS symbols the same public signature. However, Universo is free to chose the private signature to use. Since the private signature contains in practice reduction rules, this means that Universo needs to control the reduction. This will be detailed in Section 8.3. Also, we have designed Universo so that each of the 4 steps mentionned previously can be computed separately and parameterized via a configuration file. But not only, even the solver used for the third step could be reimplemented: Either by calling another SMT solver or by using an ad-hoc algorithm (as for Coq). We think that having this design is important, since in practice, we may face scalability issues for large libraries. The parameterization of Universo for the arithmetic library of Matita will be detailed in Chapter 11.

This Chapter is organized as follows: In Section 10.1 we give a high-level description of Universo and how it works. In Section 10.2 we present how Universo can be parameterized by the user. In Section 10.3 we give implementation details of Universo. In particular we will show how it is built around Dedukti's kernel. In Section 10.4, we explore several potential extensions of Universo.

10.1 Universo in a nutshell

Universo is a tool for Dedukti that implements the algorithm described in Chapter 2 which addresses the following problem: Given a derivable judgment $\Gamma \vdash_{\mathscr{C}} t : A$ in a CTS $\mathscr{C}$, is it possible to embed this judgment in $\mathscr{C}'$ in a way that $\Gamma' \vdash_{\mathscr{C}'} t' : A'$ is derivable, where $\Gamma =_\star \Gamma', t =_\star t'$ and $A =_\star A'$. The equality $=_\star$ equates two terms (and is extended to contexts) if they are equal modulo the sorts (Definition 13). In Section 2.3, we have described an algorithm deciding this problem, and Universo is an implementation of this algorithm. In particular:

- Universo uses Dedukti as a type checker. Hence Universo uses the embedding of CTS into λΠ-calculus modulo theory we saw in Chapter 6 and its implementation for Dedukti presented in 8.3. The main issue while working through this encoding is the generation of the free CTS. In Section 10.3, we explain how the generation of a free CTS is done in Universo.

- In practice, the system being encoded often has other features such as inductive types. Universo needs to be compatible with these features. The fact that Dedukti is a very weak language makes this task easy. The main issue is about rewrite rules, this is detailed in Section 10.3.4.

- Proofs are not presented as typing judgments but as files, hence the environment such as the namespace system of Dedukti needs to be taken into account in practice.

- In the λΠ-calculus modulo theory (and therefore in Dedukti) subtyping is explicit. Because Universo cannot remove or add any subtyping proofs (yet?), Universo assumes the proofs has been generated with identity casts, even if they are identity casts. As discussed in 2.4, adding these identity casts does not break the soundness of Universo but help to get completeness.

A typical use of Universo is as follows: The user has a proof in some file `A.dk` written in a logic $\mathcal{L}$ where the underlying CTS is $\mathscr{C}$ and wants to translate this proof in a logic $\mathcal{L}'$ which differs from $\mathcal{L}$ only by the underlying CTS which is $\mathscr{C}'$. Also, we consider that the logics $\mathcal{L}$ $\mathcal{L}'$ are described in only one Dedukti file which we call here `cts.dk` for both $\mathcal{L}$ and $\mathcal{L}'$. The target specification $\mathscr{C}'$ is given to Universo via a configuration file described in Section 10.2. It is the responsability of the user, that the target specification written in this configuration file is the same as the underlying CTS of the logic $\mathcal{L}'$. A typical invocation of Universo with the command line is

```
universo -o out --theory logic.dk --config config.dk A.dk
```

If Universo succeeds, this command generates a file `out/A.dk`. If the target specification given by the user is compatible with the one in `logic2.dk`, then the file generated by Universo is well-typed. In practice, a library is split among several files in several directories, this is also supported by Universo see 10.2. However, Universo does not handle dependencies between

```
def example : cts.Univ cts.s2 :=
  cts.prod cts.s2 cts.s2 cts.s2 cts.I
    (cts.cast
       cts.sinf
       cts.sinf
       (cts.univ cts.s2 cts.sinf cts.I)
       (cts.univ cts.s2 cts.sinf cts.I)
       cts.I
       (cts.univ cts.s1 cts.s2 cts.I))
    (__ =>
     cts.cast
       cts.sinf
       cts.sinf
       (cts.univ cts.s2 cts.sinf cts.I)
       (cts.univ cts.s2 cts.sinf cts.I)
       cts.I
       (cts.univ cts.s1 cts.s2 cts.I)).
```

Figure 10.1: A proof in DEDUKTI of the judgment $\vdash_{\mathscr{D}_3} s_1 \to s_1 : s_2$ using the CTS encoding

modules. Instead, we use a usual combination between an external tool–`dkdep`[1], an external tool to compute the appropriate dependencies between modules in DEDUKTI–and a `Makefile` (similarly to `ocamldep`).

To understand how each part articulates with one another, we will use a running example which is the same as that in Example 2.13. The original judgment $\vdash_{\mathscr{D}_3} s_1 \to s_1 : s_2$ in DEDUKTI[2] is presented in Fig. 10.1.

Each following subsection gives a description of the files taken as input and the ones generated on this running example.

10.1.1 Elaboration step

- Input: `input/A.dk`
- Output: `output/A.dk`, `output/A_elab.dk`

Elaboration goes through the input file and replace all the sorts by a fresh variable. Sorts that need to be elaborated are given by the configuration file (see 10.2) as rewrite rules. The elaboration is divided into two smaller steps: First, terms are normalized with the rules given in the configuration files with DKMETA. This step generates constants `Universo.var`. Then, the elaboration goes through the term and replace each occurence of `Universo.var` by a fresh variable such as `A_elab.?1`. Finally, each fresh variable variable is declared in a new file: `output/A_elab.dk` shown in Fig. 10.2.

The file `output/A.dk` is the same as `input/A.dk` where every sort is replaced with a fresh sort variable as shown in Fig. 10.2. Hence a new explicit dependency via the command `#REQUIRE`[3] has been introduced to the file `output/A_elab.dk`. This is to facilitate the separation between

[1] `dkdep` is part of the DEDUKTI tool suite.
[2] Actually we translate the judgment $\vdash_{\mathscr{D}_3} s_1 \to s_1 \Rightarrow s_2$.
[3] The command `REQUIRE` makes a dependency in DEDUKTI explicit but it is not mandatory.

```
(; output/A_elab.dk ;)

(; example ;)
def ?0 : cts.Sort.
def ?1 : cts.Sort.
def ?2 : cts.Sort.
def ?3 : cts.Sort.
def ?4 : cts.Sort.
def ?5 : cts.Sort.
def ?6 : cts.Sort.
def ?7 : cts.Sort.
def ?8 : cts.Sort.
def ?9 : cts.Sort.
def ?10 : cts.Sort.
def ?11 : cts.Sort.
```

```
(; output/A.dk ;)

#REQUIRE A_elab.
#REQUIRE A_sol.

def example :
  cts.Univ A_elab.?0
  :=
  cts.prod
    A_elab.?1
    A_elab.?2
    A_elab.?3
    cts.I
    (cts.cast
       cts.sinf
       cts.sinf
       (cts.univ A_elab.?4 cts.sinf cts.I)
       (cts.univ A_elab.?5 cts.sinf cts.I)
       cts.I
       (cts.univ A_elab.?6 A_elab.?7 cts.I))
    (__ =>
     cts.cast
       cts.sinf
       cts.sinf
       (cts.univ A_elab.?8 cts.sinf cts.I)
       (cts.univ A_elab.?9 cts.sinf cts.I)
       cts.I
       (cts.univ A_elab.?10 A_elab.?11 cts.I)).
```

Figure 10.2: Output of UNIVERSO after the elaboration step

steps so that the other steps can be run indepently without executing again the elaboration which rarely changes.[4]

10.1.2 Type checking step

- Input: `output/A.dk` `output/A_elab.dk`
- Output: `output/A_cstr.dk`

Because the file `output/A.dk` is ill-typed, UNIVERSO instruments DEDUKTI's type checker to generate the free CTS (Definition 2.3.3). This instrumentation is roughly a *hook* in the convertibility test which is triggered everytime the rewrite engine of DEDUKTI sees a sort variable (generated by UNIVERSO). This instrumentation is detailed in Section 10.3. For example, everytime the type checker sees a CTS product, a constraint is added. We see in Fig. 10.3 that the term `cts.prod A_elab.?1 A_elab.?2 A_elab.?3 cts.I` has triggered the generation of the

[4] Another explicit *require* command has been introduced to the file `output/A_sol.dk`. This file is currently empty and will be filled during the last step. This dependency is here to facilitate the integration with a Makefile.

```
#REQUIRE A.

(; output/A_cstr.dk ;)

(; example ;)
[] cts.Rule A_elab.?1 A_elab.?2 A_elab.?3 --> cts.true.
[] cts.Axiom A_elab.?4 cts.sinf --> cts.true.
[] cts.Axiom A_elab.?5 cts.sinf --> cts.true.
[] cts.Cumul A_elab.?4 A_elab.?5 --> cts.true.
[] cts.Axiom A_elab.?6 A_elab.?7 --> cts.true.
[] A_elab.?7 --> A_elab.?4.
[] A_elab.?5 --> A_elab.?1.
[] cts.Axiom A_elab.?8 cts.sinf --> cts.true.
[] cts.Axiom A_elab.?9 cts.sinf --> cts.true.
[] cts.Cumul A_elab.?8 A_elab.?9 --> cts.true.
[] cts.Axiom A_elab.?10 A_elab.?11 --> cts.true.
[] A_elab.?11 --> A_elab.?8.
[] A_elab.?9 --> A_elab.?2.
[] A_elab.?3 --> A_elab.?0.
```

Figure 10.3: Output of Universo after the type checking step

constraint `[] cts.Rule A_elab.?1 A_elab.?2 A_elab.?3 --> cts.true.`. We encode constraints in Dedukti as rewrite rules. This is an interesting feature of Universo because these constraints can be easily parsed by Dedukti or Dkmeta. But not only, for equality constraints these constraints can be used to make the type checking faster!

10.1.3 Solving Step

- Input: `output/A_cstr.dk`
- Output: `output/A_sol.dk`

The constraints generated in the file `output/A_cstr.dk` can be given to a solver. In the case of Universo we have used the SMT solver Z3. For this example, Z3 found a solution and Universo uses this solution to generate the file `output/A_sol.dk` shown in Fig. 10.4. Again, we use rewrite rules to write this solution, so that they can be processed with Dkmeta later.

At the end of this step, the file `output/A.dk` can be type checked in the target specification.

10.1.4 Reconstruction

- Input: `output/A.dk output/A_sol.dk`
- Output: `output_reconstruction/A.dk`

This step is not mandatory to hav typ checkabe files but it makes Universo easier to use with other tools as presented in Chapter 11. This step generates a file where every sort variable introduced by Universo has been replaced with the solution found in the previous

```
#REQUIRE A_elab.

(; output/A_sol.dk ;)

[] A_elab.?0 --> cts.s3.
[] A_elab.?1 --> cts.s3.
[] A_elab.?2 --> cts.s3.
[] A_elab.?3 --> cts.s3.
[] A_elab.?4 --> cts.s2.
[] A_elab.?5 --> cts.s3.
[] A_elab.?6 --> cts.s1.
[] A_elab.?7 --> cts.s2.
[] A_elab.?8 --> cts.s2.
[] A_elab.?9 --> cts.s3.
[] A_elab.?10 --> cts.s1.
[] A_elab.?11 --> cts.s2.
```

Figure 10.4: Output of UNIVERSO after the solving step

```
def ex : cts.Univ cts.s3 :=
  cts.prod
    cts.s3
    cts.s3
    cts.s3
    cts.I
    (cts.cast
       cts.sinf
       cts.sinf
       (cts.univ cts.s2 cts.sinf cts.I)
       (cts.univ cts.s3 cts.sinf cts.I)
       cts.I
       (cts.univ cts.s1 cts.s2 cts.I))
    (__ =>
     cts.cast
       cts.sinf
       cts.sinf
       (cts.univ cts.s2 cts.sinf cts.I)
       (cts.univ cts.s3 cts.sinf cts.I)
       cts.I
       (cts.univ cts.s1 cts.s2 cts.I)).
```

Figure 10.5: Output of UNIVERSO after the reconstruction step

step. In particular, this step removes the intermediate generated files `A_elab.dk`, `A_cstr.dk` and `A_sol.dk`. The final result is pictured in Fig. 10.5.

We see that the solution generated is the same as the one presented in Example 2.13.

10.2 Parameterization of Universo

Universo needs to be configured because it is intended to be used with many logics and many CTS specification for different purposes, but also for scalability. Besides the options given to Universo via the command-line, one needs to parameterize Universo with an external configuration file. We refer the reader to the documentation of Universo to an exhaustive presentation of Universo's command line options and parameterization. Below, we explain how the parameterization solves some common problems.

The syntax of CTS differs from the sorts. This is an issue for Universo because it needs to handle sorts for any CTS in the same way. Because in practice all the sorts are countable, Universo uses an internal representation for sorts isomorphic to natural numbers. These constants are `uzero` and `usucc`. We use the constant `enum` to go from natural numbers to sorts. Notice that these constants are purely syntactic for Universo. The user will give a semantics to these constants in the configuration file.

In Universo, the configuration file is actually a Dedukti file. Dedukti's syntax offers a simple way to write a configuration file to parse and at the same time may contain valid rewrite rules that can be used by Dkmeta. The configuration file of Universo is split into different unordered sections. A section is introduced with a definable declaration of type `Type` in Dedukti syntax.

In particular, Universo recognizes 4 sections:

- `elaboration`: Configure the elaboration step
- `constraints`: Add additional constraints to Universo
- `solver`: Configure the solver used by Universo
- `output`: Configure the output of Universo

Elaboration: This section contains rewrite rules which will be used as meta rewrite rules (with Dkmeta) to replace every sort by a fresh variable. For example one can write:

```
1 [] cts.star --> cts.var.
```

Everytime a term matches the left pattern it will be replaced with the constant `cts.var`. Then, Universo automatically replaces this variable by a fresh variable. The user can also map a sort to the internal representation of Universo sorts (internal representation of sorts is detailed in the output paragraph).

```
1 [] cts.star --> cts.enum cts.uzero.
```

This feature is interesting because most of the time a sort such as $\star$ in many CTS specification never needs to be changed because it reprensents a proposition and is at the bottom of the universe hierarchy. Notice that this optimization means that Universo can do more than generating the free CTS. A side-effect of using Dkmeta is that this section can contain arbitrary meta rewrite rules. This interesting feature can be used to preprocess files. In particular, it may speed up the solving step if the precomputation reduces the number of constraints that will be generated during type checking.

Constraints: This section allows the addition of constraints by the user. The reason is that in practice, UNIVERSO the specification morphism is not unique from the free CTS to the target specifcation. For example, in Chapter 11, we explain that by default, UNIVERSO maps the natural numbers defined in the MATITA's arithmetic library as a proposition $\star$ and not as a datatype $\Box$. In general, people prefer to use natural numbers as a datatype. To prevent UNIVERSO to sort natural numbers as proposition, the user can add additional constraints such as:

```
[] matita_arithmetics_nat.nat --> cts.Cumul (cts.enum (cts.usucc cts.uzero))
↪  cts.var.
```

which says that the sort for natural numbers should be *greater* than the sort assigned to `cts.enum (cts.usucc cts.uzero)`. In general, the sort `cts.enum cts.uzero` is used for proposition ($\star$).

Solver: This section contains several options to parameterize the solver. Currently the only solver implemented for UNIVERSO is Z3. In UNIVERSO, Z3 can be used with two different logics which are:

- `QF_UF` (Quantifier Free Uninterpreted Function symbols): An extension of propositional logic with equality and non interpreted functions symbols
- `LIA` (Linear integer arithmetic): Every variable is interpreted as an integer. Linear means that there is no multiplication.

The semantics to the constants `uzero` and `usucc` depends on the logic. In the case of `LIA`, `uzero` denotes 0 and `usucc` denotes the successor operator. In the case of `QF_UF`, these are just symbols for which their interpretation is given by the user. In both cases, the user needs to provide an interpretation.

Configuring UNIVERSO with the `LIA` logic: This section only needs to be specified if the user uses Z3 with the `LIA` logic. In this section, the user has to define three symbols: `axiom a b`, `rule a b c` and `cumul a b`. The definition is given via the DEDUKTI syntax for rewrite rules. The right-hand side of the rewrite rule should be a term over the following algebra (a symbol and its arity):`true` (0), `false` (0), `zero` (0), `succ` (1), `eq` (2), `max` (2), `imax` (2), `le` (2), `ite` (3). We will only detail the interpretation associated to `imax`, for the other symbols, the name should be self-explanatory or can be found in [BST10]. `imax` encodes an impredicative `max`, it could be defined with the other symbols as:

```
  [a,b] imax a b --> ite (b zero) zero (max a b).
```

We have `imax` in the algebra because in practice it is often used. A set of valid rewrite rules for this section could be for example:

```
[a,b]   axiom a b   --> eq (succ a) b.
[a,b,c] rule  a b c --> eq (imax a b) c.
[a,b]   cumul a b   --> le a b.
```

The specification encoded by these rules is an impredicative hierarchy of universes as defined by the CTS of LEAN in Definition 1.5.13. Notice that while the `LIA` is well-suited for CTS behind LEAN, MATITA or COQ, it is not that convenient pas the CTS specification of STT$\forall$ for example. While $\star$ and $\Box$ can be associated to the numbers 0 and 1 it is not clear what number

should be associated to the sorts $\triangle$ and $\Diamond$ for example. But also, it requires more complex interpretations for symbols `axiom` or `rule`. For this reason, when the specification uses a finite number of sorts, it may be better to use the `QF_UF` logic.

Configuring Universo with the `QF_UF` logic: As for `LIA`, this section only needs to be specified if the user uses Z3 with the QF_UF logic. In this section, the user needs to give an exhaustive interpretation for $\mathcal{A}$, $\mathcal{R}$, and CC. Hence, QF_UF can be used only for finite specifications. This is not a real restriction since a proof uses only a finite number of universes. However, this requires guessing the maximum number of universes used by those proofs. In practice, this number is rather low and rarely exceeds 5.

The user only needs to specify an interpretation when the relation is true. An example of specification for this section could be:

```
[]  cts.Axiom star box       --> cts.true.

[]  cts.Rule star star star  --> cts.true.
[]  cts.Rule star box  box   --> cts.true.
[]  cts.Rule box  star star  --> cts.true.
[]  cts.Rule box  box  box   --> cts.true.

[a] cts.Cumul a a            --> cts.true.
```

which encoded the specification of Calculus of Constructions. Notice that `cts.Cumul` is not reflexive by default.

Output: This section is used by Universo to map its own representation of universes to the ones of the target specification. This is achieved using meta rewrite rules.

```
[]  cts.enum cts.uzero              --> star.
[]  cts.enum (cts.usucc cts.uzero)  --> box.
```

This section is used by the `LIA` logic to interpret the solution of the solver: A variable mapped to the number 0 by the solver will be interpreted by `star` using the rules above. When using the `QF_UF` logic, the solver map a variable to `cts.enum cts.uzero` or `cts.enum (cts.usucc cts.uzero)` directly. Hence the rules of this section are just composed with the output of the solver.

10.3 Implementation of Universo

In this section we explain several design choices that have been made for Universo. For scalability, Universo uses many tricks related to the rewrite engine of Dedukti. We hope that this section may enlighten a person aiming to further work on Universo.

Once the terms have been elaborated, they are not well-typed in the source logic. The purpose of the type checking step is to generate constraints so that the terms are well-typed under these constraints. These constraints represent the free CTS of the derivation tree computed by Dedukti as presented in Section 2.3. We insist that these constraints–a priori–do not depend only of the proof, but also on the type checker. It is not clear whether if we change the type checker, the constraints generated encodes the same free CTS (or a free CTS equivalent). This is why our method is not complete. In particular, considering performance we do not compute the SNF of a term but only its WHNF(Definition 8.1.2).

We may observe that this procedure is really similar to what is done in Coq with floating universes. Except that the algebra of constraints for Coq is fixed and hence, an ad-hoc algorithm [Typ05] could be implemented for this specification.

Since the elaborated term is ill-typed, invoking Dedukti's type checker on it fails. To overcome this issue, Universo implements a hook over the convertibility test of Dedukti. In OCaml, this is done via a functor mechanism that is presented in Section 8.1.4. The purpose of this hook is to catch cases that involve the fresh sorts elaborated at the previous step. As a consequence, this step depends crucially on the private encoding of CTS because Universo needs to know the shape of a term in WHNF in the CTS encoding. The private signatured used for Universo is the one presented in Section 8.3.

The hook of Universo comes before the convertibility test of Dedukti. It takes two terms in WHNF and returns a boolean: `true` meaning that these two terms are convertible for Universo. This means that as a side effect a constraint has been generated. `false` meaning that Universo does not know whether these two terms are convertible and the usual convertibility test of Dedukti takes over.

The hook of Universo implements eight cases that can be divided by two because of symmetry. Given two terms `l` and `r` in WHNF, checks the following cases:

- If `l` = `?i` and `r` = `?j` then we add the constraint `?i` = `?j`
- If `l` = `Rule ?i ?j ?k` and `r` = `true` then we add the constraint `Rule ?i ?j ?k` = `true`
- If `l` = `Axiom ?i ?j` and `r` = `true` then we add the constraint `Axiom ?i ?j` = `true`
- If `l` = `Cumul ?i ?j` and `r` = `true` then we add the constraint `Cumul ?i ?j` = `true`

The first constraint generates the equivalence relation for the free CTS. The other constraints encode the specification of the free CTS.

As an optimization, the first constraint is also added as a rewrite rule in Universo. This optimization speeds up the type checking when proofs are getting big. Adding this rewrite rule should be done with care: Indeed, a wrong orientation of the constraint as a rewrite rule may introduce a non-terminating rewrite system. We avoid this by implementing a total order on the sorts elaborated by Universo. This order corresponds to the underlying order of natural numbers. Hence `?5` > `?2`. An equality is always oriented from the larger universe to the smaller one. We have made this choice because empirically, fewer constraints are added on smaller universes. Hence, this heuristic makes the computation of the WHNF of a universe faster.

10.3.1 Identity casts and non-linearity

The non-linear rule coming from identity casts is a big issue. In practice, identity casts allows a faster type checking and seems necessary[5] with the current encoding of inductive types in the λΠ-calculus modulo theory. This means that without the canonicity rule (in the private signature)

```
1 [A,t] cast' _ _ A A t -> t.
```

the type checking of a term may fail. An example of this the following one (using the CTS encoding in Dedukti)[6]:

[5] We recall that in Section 6.4 we already discussed about the necessity of identity casts.
[6] We have applied some reduction rules to make the example easily readable.

```
def u      := cts.univ star box cts.I.
def v      := cts.univ star box cts.I.
def cast   := cts.cast box box u v cts.I.

A : cts.Univ star.

f : cts.Term star (cast A).

eq : A : cts.Univ star ->
     cts.Term star A ->
     cts.Term star A ->
     cts.Univ star.

#INFER (eq A f f). (; cts.Univ str ;)
```

The term `eq A f f` thanks to the identity cast is well-typed. If we use UNIVERSO[7] on this example, we will obtain the following term

```
def u      := cts.univ ?1 ?2 cts.I.
def v      := cts.univ ?3 ?4
def cast   := cts.cast ?5 ?6 u v cts.I.

A : cts.Univ ?7.

f : cts.Term ?8 (cast A).

eq : A : cts.Univ ?9 ->
     cts.Term ?10 A ->
     cts.Term ?11 A ->
     cts.Univ ?12.

#INFER (eq A f f). (; Ill-typed without constraints ;)
```

To type check the term `eq A f f`, DEDUKTI's type checker checks that `cast A` is convertible to `A`. To do so, DEDUKTI's type checker computes the WHNF of `cast A`. The rewrite engine first unfolds the definition of `cast` to get the term `cts.cast ?3 ?4 u0 u0 cts.I A`. At this stage, since `cts.cast` is a definable symbol, to compute the WHNF, DEDUKTI's type checker tries to see whether there is a rule which matches against this term. One such rule is the identity cast rule. But to know whether this rule matches, it needs to know whether the sort variable `?1` (resp. `?2`) is convertible to the sort variables `?3` (resp. `?4`). To do so, it calls the convertibility test (hooked by UNIVERSO). In that case, the hook of UNIVERSO says yes and add two constraints. Therefore, DEDUKTI's type checker can apply the identity cast rule and says that the term is well-typed.

But there is a catch here. It means that whenever DEDUKTI's type checker tries to apply an identity cast, it succeeds and this is not desired!

For example, if we change the example by the one below

```
def U0 := cts.Univ star.
def U1 := cts.Univ box.
```

[7] The type checking phase we are discussing here is independent from the target specification.

```
3  def u0 := cts.univ star box cts.I.
4  def u1 := cts.univ box triangle cts.I.
5
6  B : U0.
7  g : U1 -> U0.
8  x : cts.Term star (g u0).
9  y : cts.Term star (g (cts.cast box triangle u0 u1 cts.I B)).
10 z : cts.Term box ((x : cts.Term star (g (cts.cast box triangle u0 u1 cts.I B))
   ↪  => u0) y).
```

After the elaboration step, we get

```
1  def U0 := cts.Univ ?1.
2  def U1 := cts.Univ ?2.
3  def u0 := cts.univ ?3 ?4 cts.I.
4  def u1 := cts.univ ?5 ?6 cts.I.
5
6  B : U0.
7  g : U1 -> U0.
8  x : cts.Term ?7 (g u0).
9  y : cts.Term ?8 (g (cts.cast ?9 ?10 u0 u1 cts.I B)).
10 z : cts.Term ?11 ((x : cts.Term ?12 (g (cts.cast ?13 ?14 u0 u1 cts.I B)) => u0)
   ↪  y).
```

After checking the type of the variable `y`, UNIVERSO has generated at least the following constraints

```
1  [] cts.Axiom ?3 ?4 --> cts.true.
2  [] ?5              --> ?4.
```

During the type checking of the application in the type of the variable `z`, DEDUKTI's type checker checks whether the type of the variable `y` is convertible with `cts.Term ?12 (g (cts.cast ?13 ?14 u0 u1 cts.I B))`. This comes back to checking the convertibility between `cts.cast ?9 ?10 u0 u1 cts.I B` and `cts.cast ?13 ?14 u0 u1 cts.I B`. At this stage, DEDUKTI's tries to compute the WHNF of these terms and will tries to use the identity cast rule. If it succeeds, it will generate the constraint

```
1  [] ?4              --> ?3.
```

This constraint is problematic since it means that the CTS behind COQ (limited to three universes) is not a solution[8]. To avoid this problem, the identity cast rule is removed from the private signature by UNIVERSO, and the hook it implements applies manually identity casts only when it is necessary. Meaning that the rule is applied manually only when the convertibility test of DEDUKTI's type checker checks if a cast-term is convertible with a non-cast term. We are not very pleased with this solution but it works. Morever, this trick is necessary to make UNIVERSO's work in practice.

[8] Notice that this constraint shows that the derivation tree built by DEDUKTI is important for completeness.

10.3.2 Modularity with Universo

For each file, the type checking step of Universo produces a Dedukti file which contains the generated constraints as rewrite rules. One advantage of this, is that these constraints can be reused for the type checking of other modules.

While we observe that the type checking with Universo is not much longer than the original file, we observed that importing dependencies took a lot of time which was due to the construction of the decision tree (Section 8.1.3). Indeed, the old behavior was to construct the decision tree every time a new rule was added on a symbol. With Universo, many rules are added one by one, and the fact that the decision tree is built for every rule leads to a squared complexity with respect to the number of rules. We solve this issue by making the computation of decision trees lazy: The decision tree is computed only when it is needed, meaning when the rewrite engine needs to compute the WHNF of a symbol. In practice, this slightly slows down the type checking but it is reasonable way.

10.3.3 Solving constraints

Once all the files have been type checked, Universo calls the solver with all the constraints that have been generated. So far, we only support the Z3 SMT solver with two different logics: QF_UF and LIA. For this step, we also have implemented an optimization which is to pre-process Z3's input with union-find. We observe that in many cases, this may decrease Z3's solving time.

This step is the bottleneck to make Universo scales. We cannot bound the time it takes to an SMT solver to solve these constraints. We observe that in practice the time seems to increase linearly with the size of the proofs. We have no information about proofs that are larger than those we have processed (at the time of writing Matita's arithmetic library and part of Coq standard library), but we suspect that some work remains to be done to make Universo further scalable.

10.3.4 Compatibility of Universo

In practice, we have manipulated proofs encoded from a logic which is not only a CTS but has other features such as inductive types or recursive functions. The encoding of these features in Dedukti also use rewrite rules. As mentioned in Section 10.3, the behavior of Universo highly depends on the WHNF of a term. Empirically, we observe that the encoding of these features in Dedukti does not have an impact on Universo. This is because these rewrite rules added to encode inductive types or recursive functions (see Section 8.4) are universe polymorphic. Meaning that in a pattern, there is never a concrete universe (such as $\star$ or $\Box$).

10.4 Future Work

Partially solve the constraints: In practice, we observe that for some universes, we do not need to type check the whole library to find the final solution. If we take the equality symbol for example, this symbol is one of the first symbols defined in a library. Once it is used with the highest possible datatypes in the universe hierarchy, the sort may be fixed once and for all. Fixing such a sort may ease the solver to find a solution. For the moment, the only way to do this is manually by adding more and more constraints to the `constraint` section of the configuration file of Universo. However, it would be interesting to have a mechanism for producing partially specified solution when given a part of the library. By splitting the library into parts and process them separately, we suspect that this may decrease the solving time a lot.

Non-linear rules: The solution we have proposed for the non-linear rules described in Section 10.3 requires a really good understanding of the encoding. We also realize that in practice the rules ordering matters because of non-linear rule may have an impact on the type checking time. It would be interesting to investigate this and see whether Dedukti could offer a better API to handle this problem in an easy way.

Instrument the SMT solver: We have made little effort to instrument the SMT solver. Two SMT-solving features may speed up the solving timee: First, most SMT-solvers have an incremental solving mode which could be used to submit and solve the constraints after each proof in the library rahter than at the end. Another feature to guide the SMT solver are tactics. Z3 proposes a set of tactics but also to implement our own tactics to guide the SMT solver. It would be interesting to see whether tactics may decrease the solving time.

Chapter 11

The Matita Arithmetic Library into STT∀

In this chapter, we show our main practical result using tools we have presented in the previous chapters Dkmeta in Chapter 9, and Universo in Chapter 10. Our result is a semi-automatic translation from proofs expressed in the encoding of Matita in Dedukti to the encoding of STT∀ in Dedukti. Since the logic behind Matita is more expressive than Higher-Order Logic, we are interested into theorems that can also be expressed in Higher-Order Logic. This is why in this work, we have only translated part of the Matita arithmetic library into Higher-Order Logic where the last theorem proved is Fermat's Little Theorem. This proof needs about 300 lemmas which makes this example large enough to require automatation. The full translation is split into three steps:

1. We translate Matita's proof into Dedukti. For this, we have used the CTS encoding we have shown in Section 8.3. The effective tool which goes from Matita to Dedukti is called Krajono[1] and has been originally designed by Ali Assaf [Ass15b] and extended for this work to take the new encoding of CTS into acccount.

2. We make several translations to go from the Matita's logic encoded in Dedukti to the encoding of STT∀ in Dedukti presented in Section 8.2. This cannot be achieved with Universo only because the logic of Matita contains inductive types and recursive functions.

3. From the encoding of STT∀ into Dedukti we can export these proofs into several systems such as Coq, Lean, PVS or OpenTheory, a language which is interoperable with the different systems of the Higher-Order Logic family. This exportation will be explored in Chapter 12.

The purpose of this chapter is to explain the second step of this process and explore the technical details which arise in practice. The implementation of this process is really similar to a compiler with several compilation steps. Here, we have a main proof transformation procedure achieved by Universo and then several small steps to remove features that are in Matita but not in STT∀. In our case, these features are dependent types, inductive types with their destructor (`match`) and the fixpoint operator. These last two features add a layer of complexity because the encoding we have presented in Section 8.4 introduces many rewrite rules as well as a bit of universe polymorphism. This chapter describes how this translation process has been

[1] https://github.com/Deducteam/Krajono

implemented even though the end of this translation is not completely automatized yet at the time of writing.

This chapter is organized as follows: In Section 11.1, we give an overview of Fermat's little theorem proof as it is implemented in the arithmetic library of Matita. In Section 11.2, we prune the arithmetic library to keep only what is necessary to prove Fermat's little theorem using the tool Dkprune. In Section 11.3, we explain how we used Universo to embed the proof into an extension of STT∀ which is enriched with dependent types. In Section 11.4, we use another tool called Dkpsuler to remove the polymorphic constants `match`. In Section 11.5, we remove the dependent types that come from inductive types with Dkmeta. Finally, in Section 11.6, we remove the rewrite rules which come from inductive types and recursive functions.

11.1 Fermat's little theorem and its proof in Matita

In this section, we present a proof of Fermat's little theorem. First, as an informal proof which is easy to understand and then, we give a brief description of Matita's arithmetic library that was used to implement this proof in Matita.

11.1.1 Small analysis of the proof of Fermat's little theorem

Fermat's little theorem has two equivalent statements. The one we will prove is formalized below.

Theorem 11.1.1 (Fermat's little theorem) *For all prime number p and natural number a, if p does not divde a, then $a^{p-1} \equiv 1 \mod p$.*

Proof *Considering the following equality:*

$$(p-1)! \times a^{p-1} = \prod_{i=0}^{p-1} i \times a \tag{11.1}$$

Because p is prime and does not divde a then

$$(\prod_{i=0}^{p-1} i \times a) \equiv (\prod_{i=0}^{p-1} i) \mod p \tag{11.2}$$

Hence we can conclude

$$\begin{aligned}
(p-1)! \times a^{p-1} &\equiv \prod_{i=0}^{p-1} i \times a \mod p && \text{by 11.1}\\
(p-1)! \times a^{p-1} &\equiv \prod_{i=0}^{p-1} i \mod p && \text{by 11.2}\\
(p-1)! \times a^{p-1} &\equiv (p-1)! \mod p\\
(p-1)! \times \left(a^{p-1} - 1\right) &\equiv 0 \mod p\\
a^{p-1} - 1 &\equiv 0 \mod p && \text{by Euclid's lemma}\\
a^{p-1} &\equiv 1 \mod p
\end{aligned}$$

11.1.2 Description of Matita's arithmetic library to prove Fermat's little theorem

The arithmetic library of Matita was imported from Coq's arithmetic library and is split into three directories:

- `basics` (17 files) which contains common definitions which are shared with other libraries such as Leibniz equality (as an inductive type), usual connectives, boolean type...
- `arithmetics` (26 files) which contains the main arithmetic definitions of the library.
- `arithmetics/chebyshev` (6) files) which contains results about Chebyshev polynomials and have a proof of Bertrand's theorem.

For Fermat's little theorem (named `congruent_exp_pred_S0` in the library), only the first two directories are used. Even if all these files are not used to prove Fermat's little theorem, we import all these files into Dedukti. We prefer to do it this way and prune the unecessary theorems in Dedukti because pruning the unecessary theorems is already related to our interoperability task.

The informal proof presented above requires the notion of product, factorial, congruence and permutation (to prove 11.2). Since the library has not been designed around this theorem, the definition of product is an instantiation of a more general definition called `bigops` which aims to define operators such as $\sum$ or $\prod$.

The proof of Fermat's little theorem as it is in Matita's arithmetic library cannot be exported directly in Higher-Order Logic for the following three reasons:

1. `bigops` definition is polymorphic but not in the prenex polymorphism fragment. The first argument is a natural number.
2. A second problem comes from the encoding of inductive types which are encoded with a constant `match`. However, this constant in Dedukti needs to be made universe polymorphic (as explained in Section 8.4.2). We could remove the universe polymorphism by instantiation and then use Universo. However, we realized that it is better to use Universo first, and do this instantiation later. The reason is that universe polymorphism gives more flexibility to Universo. Removing universe polymorphism introduced by the `match` is equivalent to making more sorts in the free CTS (Definition 2.3.3) equal and therefore this reduces the search space of Universo. This problem is similar to the completeness issue of Universo we already mentioned in Section 2.4.
3. A third problem is also related to the encoding of inductive types where the induction principle comes naturally with dependent types which, in the case of Fermat's little theorem, are never used. For example, one needs dependent types to type check $\mathbb{N} : \mathfrak{I} \vdash_{\mathscr{C}} \lambda x:\mathbb{N}.\mathbb{N} : (x:\mathbb{N}) \to \mathfrak{I}$, but because x is not used, we could have the following judgment instead $\mathbb{N} : \mathfrak{I} \vdash_{\mathscr{C}} \mathbb{N} : \mathfrak{I}$.

In this chapter, we will not tackle the first problem and will assume that the original definition of `bigops` is in fact in STT$\forall$. The reason why the definition is not in STT$\forall$ is simply because the quantification over a type is not prenex. Since the first arguments do not depend on this type, the definition can be made prenex polymorphic by moving this type argument to the front. We will discuss in 11.7 how this permutation could be handled directly in Dedukti.

The second issue is tackled in Section 11.4 and the third issue in Section 11.5.

```
#GDT matita_arithmetics_fermat_little_theorem.congruent_exp_pred_S0.
```

Figure 11.1: Configuration file for Dkprune

11.2 Step 1: Pruning the library with Dkprune

Pruning the library is done with a tool called Dkprune[2]. This tool has been implemented as a tool for Dedukti in OCaml. It has around 200 lines of code. It takes a configuration file as input and a directory containing a Dedukti library. A configuration file for Dkprune is a Dedukti file which can declare a sequence of identifiers (the identifier is prefixed by `#GDT`) or module name (prefixed by `#REQUIRE`) of Dedukti. Dkprune prints in an output directory the (down) transitive closure of the dependencies for all the names present in the configuration file. For this translation, we are only interested in keeping the dependencies for Fermat's little theorem, hence the configuration file of Dkprune is only one line which is presented in Fig. 11.1. Since the namespace mechanism of Dedukti is not very expressive, the file system hierarchy is not respected and the name of the folders of Matita are added to the name of the file where the theorem is declared.

Dkprune prints in the output directory specified by the user a set of non-empty files which contains only what is necessary to prove Fermat's little theorem.

11.3 Step 2: Using Universo to go to STT∀

The usage of Universo for the purpose of targetting STT∀ raised two problems we mentioned previously:

- The first issue is that which mentioned in Section 2.4 about completeness. Since some casts are missing, some rules need to be added to the specification.
- The second issue is about the encoding of inductive types with the constant `match` which is universe polymorphic.

Inductive types with Universo The destruction of an inductive type in Dedukti is done via a constant `match` specific to this inductive type. This constant is used both to construct a new statement (as a proposition in Calculus of Inductive Constructions) or to construct a new type. In the case of the inductive type $\mathbb{N}$, the type of this constant is the following:

$$(s : \mathcal{S}) \to (P : \mathbb{N} \to s) \to P\ 0 \to ((x : \mathbb{N}) \to P\ (n+1)) \to (z : \mathbb{N}) \to P\ z$$

There is a quantification on a sort s because the constant `match` is universe polymorphic (see Section 8.4). In the library, this constant is used both with a sort $\star$ for proposition and $\Box$ for types. However, in this last case, this leads to the use of a dependent type. In STT∀, the induction principle can be stated as it is, but one cannot define new objects with this definition because it requires to use a dependent type which does not exist in STT∀. Eliminating universe polymorphism (a.k.a eliminating the quantification over a sort) introduces a duplication for this constant. However, it is better to run Universo before this duplication as discussed previously in Section 11.1.1.

[2] https://github.com/Deducteam/Dedukti/blob/master/commands/dkprune.ml

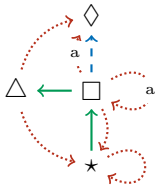


Figure 11.2: CTS graph used with UNIVERSO to translate MATITA's arithmetic library

Since the type of the constant `match` is not in the CTS encoding, this use of a dependent type is not seen by UNIVERSO because the type is already in a reduced form. Hence there is no need to add a dependent type in the configuration file even if implicitly there are dependent types in the library.

The specification In Fig 11.2, we present the difference between the STT∀ specification and the one we use in practice. There are mainly two differences: The first one is the addition of a rule

```
1 [] cic.Rule cic.box      cic.box      cic.diamond  --> cic.true
```

which is derivable in STT∀ because $(\Box, \Diamond) \in \mathcal{C}_{\text{STT}\forall}$ and $(\Box, \Box, \Box) \in \mathcal{R}_{\text{STT}\forall}$. In practice, since we have explicit casts it makes a difference to allow the former rule. The second difference is that because all the type operators are of arity 0 we don't need the rule for type operators and so the sort $\circ$ is not needed too.

Finally, if we had a real encoding of inductive types in DEDUKTI, one would need to add dependent types with the rule $(\star, \star, \Box)$ but for reasons explained previously we do not need to add this rule in the specification.

The configuration file for UNIVERSO is given in Fig 11.3.

11.4 Step 3: Removing universe polymorphism of inductive types

This step removes the universe polymorphism introduced by the constants `match` and `filter` used to encode inductive types. An invariant of the specification we have given to UNIVERSO is that these universe polymorphic constants can be applied only to the sorts $\star$ and $\Box$ (or `cic.star` and `cic.box`). This instantion is done via another tool: DKPSULER. DKPSULER aims to be a more general tool that duplicates symbols. The idea behind this tool is to duplicate a symbol whenever it matches some pattern. Everytime a symbol matches a pattern, this pattern is replaced by a new symbol provided by the user. The user may provide the patterns and the symbols simply using a rewrite rule.

11.4.1 DKPSULER

As for the tools we presented so far, DKPSULER takes a configuration file in parameter which contains rewrite rules of the form

```
1 [] f a --> g.
```

which means that every time the symbol `f` is applied to the symbol `a`, it will be replaced by the new symbol `g`. DKPSULER creates the symbol `g` as a DEDUKTI declaration or a definition depending whether f is itself a declaration or a definition. The type of `g` is the same as `f a`.

```
def elaboration : Type.
    (; [] cic.prop    --> cic.var. ;)
    []  cic.prop   --> cic.enum cic.uzero.
    [s] cic.type s --> cic.var.
    [s] cic.succ s --> cic.var.

def output : Type.
    []  cic.enum cic.uzero                                        --> cic.star.
    []  cic.enum (cic.usucc cic.uzero)                            --> cic.box.
    []  cic.enum (cic.usucc (cic.usucc cic.uzero))                -->
    ↪  cic.triangle.
    []  cic.enum (cic.usucc (cic.usucc (cic.usucc cic.uzero))) --> cic.diamond.

def constraints : Type.
    [] matita_arithmetics_nat.nat     --> cic.Cumul (cic.enum (cic.usucc
    ↪  cic.uzero)) cic.var.
    [] matita_basics_bool.bool        --> cic.Cumul (cic.enum (cic.usucc
    ↪  cic.uzero)) cic.var.
    [] matita_basics_lists_list.list  --> cic.Cumul (cic.enum (cic.usucc
    ↪  cic.uzero)) cic.var.

def solver : Type.
    [] solver  --> z3.
    [] logic   --> qfuf.
    [] opt     --> uf.
    [] minimum --> 4.
    [] maximum --> 4.
    [] print   --> true.


def qfuf_specification : Type.

   [] cic.Axiom cic.box cic.triangle      --> cic.true.
   [] cic.Axiom cic.star cic.box          --> cic.true.
   [] cic.Axiom cic.diamond cic.sinf      --> cic.true.
   [] cic.Axiom cic.triangle cic.sinf     --> cic.true.
   [] cic.Rule cic.star     cic.star     cic.star     --> cic.true.
   [] cic.Rule cic.box      cic.box      cic.box      --> cic.true.
   [] cic.Rule cic.box      cic.star     cic.star     --> cic.true.
   [] cic.Rule cic.triangle cic.star     cic.star     --> cic.true.
   [] cic.Rule cic.triangle cic.diamond  cic.diamond  --> cic.true.
   [] cic.Rule cic.box      cic.box      cic.diamond  --> cic.true.
   [] cic.Cumul cic.box  cic.diamond  --> cic.true.
   [a] cic.Cumul a a                  --> cic.true.

def end : Type.
```

Figure 11.3: Configuration file for UNIVERSO to go to STT∀ with dependent types

```
files=$*
regex="def match_(.*) :" # regexp to get inductive names
theory="cic"
for f in $files
do
    md=${f##*/}
    md=${md%.*}
    while read line; do
        if [[ $line =~ $regex ]]
        then
            ind=${BASH_REMATCH[1]}
            echo "[] ${md}.match_$ind $theory.star  --> ${md}.match_${ind}_star."
            echo "[] ${md}.match_$ind $theory.box   --> ${md}.match_${ind}_box."
            echo "[] ${md}.filter_$ind $theory.star --> ${md}.filter_${ind}_star."
            echo "[] ${md}.filter_$ind $theory.box  --> ${md}.filter_${ind}_box."
        fi
    done < $f
done
```

Figure 11.4: Generation of a configuration file for Dkpsuler

11.4.2 Generation of a configuration file for Dkpsuler

To apply Dkpsuler with our proofs, we need to provide a configuration file. In our case, we need to instantiate all the universe polymorphic constants introduced by the embedding of inductive types. In particular, for every inductive types in the library we need to instantiate the `match` and `filter` constants. We have decided to generate such file via a `bash` script presented in Fig. 11.4.

This script relies of course on the encoding of inductive types made by the tool Krajono as described in Section 8.4. We post-process the output of Dkpsuler with Dkmeta and Dkprune. Dkmeta is used on the fresh constants generated by Dkpsuler to compute their canonical type (see Section 9.3.2), this way we observe the use of dependent types. Dkprune is used to remove constants which are never used (if a `match` is not applied to some specific constant)[3].

11.5 Step 4: Dependent Types

Since the encoding of universe polymorphism is too shallow, meaning that products of the original system are directly encoded as a Dedukti product and not via the constant `prod` (as mentioned in 8.4.2), the previous step has made a use of dependent types explicit which Universo missed.

Fortunately, these dependent types are never used in practice and could be removed. The idea is the following one: Every time we have a product $(x : A) \to B$ which uses the rule $(\star, \square, \square)$ we can check that x is never free in B. Hence this product can be replaced by B. Since these products come from the encoding of inductive types in Dedukti, we can check that an inhabitant of a dependent type is either a variable or an abstraction $\lambda x : A.\, t$. If it is a variable, then we can simply change its type, if it is an abstraction, one needs to check that x is not free in t. Finally, some applications may become ill-typed such as $f\ a$ when the type of f was a dependent type. In that case, we just remove the argument a.

[3]The use of Dkprunecould be avoided if we were smarter to generate the configuration file.

```
(; products ;)
[A,B,C]  ltyped.app _
        (ltyped.app _
        (ltyped.app _
        (ltyped.app _
        (ltyped.app _
        (ltyped.app _ (ltyped.sym cts.prod) (ltyped.sym cts.prop))
                                                    (ltyped.sym
                                                    ↪  cts.type))
                                            (ltyped.sym cts.type))
                                            (ltyped.sym cts.I))
                            A)
                            (cts.lam C (x => B)) --> B.

(; application ;)
[A,B,f,a] ltyped.app (cts.Term cts.type (cts.prod cts.prop cts.type cts.type
↪  cts.I A (x => B))) f a --> f.

(; abstraction ;)
[A,B,t] ltyped.lam (cts.Term cts.type (cts.prod cts.prop cts.type cts.type
↪  cts.I A (x => B))) (x => t) --> t.
```

Figure 11.5: Removing dependent types when the dependency is never used

Such computation can be done directly in Dkmeta via the Light-typed quoting function presented in Section 9.2.3. The Dkmeta file is presented in 11.5. All the symbols starting with `ltyped` are related to the quoting mechanism of Dkmeta.

The pattern of the first rule is the encoded version of `cts.prod cts.prop cts.type cts.type cts.I A (x => B)`. We can apply this meta file on the whole library and remove all the dependent types that were introduced from the previous step.

11.6 Step 5: Axiomatize Inductive Types and Recursive Functions

The last step to translate the proofs into STT∀ is to remove the rewrite rules which have been introduced by the encoding of inductive types and recursive functions. This step is on paper only. The result of this process has beem done manually for the Matita's arithmetic library. This step is currently being implemented so that it can be automated.

At this step, the proofs are expressed in the CTS of STT∀ but still use rewrite rules coming from the encoding of inductive types (see Section 8.4.2) in Dedukti.

In the CTS of STT∀, there is no rewrite rules that rewrites a type of STT∀. Such a rewrite rule could not be written in the CTS of STT∀ because the head symbol of the pattern could not be well-typed in STT∀. Moreover, we know that all the rewrite rules introduced to destruct an inductive type (with the `match` constructor) and which return a proposition are never used. The reason is that such a rewrite rule maps a proof to a proof but a proof cannot appear inside a term of STT∀.

Hence, we can conclude that:

- Rewrite rules from recursive functions and inductive types are only used to type check a proof in the CTS of STT∀.
- All the rewrite rules where the left and right hand-sides are typed by a proposition in the CTS of STT∀ are never used.

If the first case, we replace the rewrite rule by an equality and replace any rewrite step by an elimination of equality. In the second case, we can simply remove this rewrite rule.

This informal argument enables us to say that any rewrite rules can be either axiomatized or removed, thus we can conclude that this step is actually always a total function. We will not detail the implementation but what we can say is that removing a rewrite and replace rewrite steps by an elimination of an equality is really similar to computing a trace. Our plan is to use Dkmeta as discussed in Section 9.3.4.

11.7 Future Work

Obviously, it would be interesting to finish the automatation of this whole process, especially the last step. But this work could be extended in several other ways and we give some ideas below.

Creating a graph of translations: We would like to reuse the different steps we have presented here in a more general way to translate a proof expressed in one logic in Dedukti into another. For this we need to have a generic way to combine the transformations we have presented in this chapter. An abstract vision of these transformations is a graph where a translation is represented as an arrow and a logic by a node. In practice, this is not so simple because we need to take into account that a development can be split into directories with dozen of files. Currently, these transformations are hard to parameterize and the combination of these transformations is done using the language `make` which is not that convenient. For example, this requires writing a `Makefile` manually but also the piece of code that combines these transformations. The fact that most of these translations need to be parameterized by a configuration file implies that the graph is actually a multigraph. The question is how the user could generate these translations and parameterize them. As it is done today, it is assumed that the user is a Dedukti expert, but as we will see in Chapter 12, it would be interesting to give a way to a non-expert to program his own translations.

Scaling these translations: A question is whether this automatation process can scale to bigger libraries such as AFP or mathematical component. We see mainly two bottle necks:

- Universo delegates its constraint problem to Z3. While we see that Z3 with Universo takes less than 1 second to solve the constraints, we may wonder what will happen for larger and larger libraries. Fortunately, the size of the problem grows linearly with respect to the size of the library, however it is not clear that it goes the same way for the solving time. If the solving time takes too much time, probably it would be reasonable to split the problem in several parts. For example, the equality is the first constant defined in the library. This equality takes a type as parameter. The sort of this type is elaborated as a variable in Universo. If one analyses the problem generated by Universo for the Fermat little theorem, this sort appears everywhere.
- The last step that has not been automated yet would not scale up as it is described today. If type checking takes n seconds, it would require at least $n \times r$ steps where r is the number of rewrite rules to be removed. One could be smarter by not removing rules one by one but

all of them at once. This is doable when we remove rewrite rules coming from inductive types and recursive functions because there are no critical pairs. Hence, everytime there is an error in type checking, it is easy to know which axiom should be used.

- Through the translations, a same theorem is type checked many times. It might be interesting to see whether this could be avoided. For example, when a translation is only local to a theorem and does not depend on the whole library like with UNIVERSO

The bigops issue: The automatation of the translation of Fermat's little theorem from MATITA to STT∀ required first to change a definition in MATITA's library so that the type of `bigop` can be encoded into STT∀. However, this is not reasonable in general because it requires manually changing the theorem before any translations. Instead, one may try to invent a transformation which changes the type of `bigop` so that it can be encoded into STT∀. In this case, it requires permuting some arguments. However, this is tricky because `bigop` is a recursive function and permuting arguments which are current encoding functions is not easy because of the `filter` functions.

Pruning on the fly: One disadvantage of our procedure is that we need to know in advance which part of the library can be translated into STT∀. It would be interesting to have a fully automated translation where theorems that cannot be translated into STT∀ are removed automatically. One difficulty lies with UNIVERSO. UNIVERSO is not able to say which theorems can be translated into one specification. Actually, the question is even harder because of the following case. Given the proof of three theorems A, B and C such that the proofs of B and C depend on A. We could have that the proofs of A and B can be encoded into STT∀ as well as A and C. But the proofs of A and B and C cannot be encoded together in STT∀.

Chapter 12

Logipedia: An Encyclopedia of Proofs

In this chapter, we discuss a concrete application that is a direct consequence of the translation we have presented in Chapter 11. This translation enabled us to express part of the arithmetic library of Matita into STT∀. STT∀ is a subset of the logic implemented by many systems today. Hence, STT∀ proofs can be easily translated in many systems. In this chapter, we show how we have exported STT∀ proofs to five different systems: Coq, Lean, Matita, PVS and OpenTheory. We choose OpenTheory because the system appears to be interoperable with many systems of the Higher-Order Logic family such as: HOL-Light, HOL4, Isabelle/HOL. Since there exists a specification morphism (Definition 2.1.1) from the underlying CTS of STT∀ to the underlying CTS of Coq, Matita and Lean, the translation is direct. For PVS, the difficulty comes from the fact that this system does not have proof terms but tactics because PVS has been designed to interact with a human. The downside of this for us is that the tactics of PVS have a sophisticated behavior, making the translation dfficult to define. Finally, for OpenTheory the translation is not as easy as we may think because this system does not have any conversion. Hence, every computation step needs to be elaborated as an equality step. Moreover, as OpenTheory does not have proof terms, it has its own sharing mechanism which is a problem in practice.

We made these translations available to any user of one of these systems with a website interface called Logipedia[1]. The main feature of Logipedia is to export a proof in a source system into a correct proof in the target system. Moreover, the website delivers theorems which are readable and could have been written in the target system directly. However, the proofs exported are not ready to be used because through the translations in Dedukti, the *structure* of the proof in the library was lost and many axioms were added. What we call *structure* is all the features offered by the proof system at the user level that the kernel (the typing system) does not see (implicit parameters, notations, coercions, ...). Logipedia exports the proof of Fermat's little theorem, with about 40 parameters and 80 axioms. The fact that our translation creates many axioms is double-edged sword. It is an advantage because it allows abstracting the representation of datatypes. For example exporting the proofs from STT∀ to Matita allows unplugging the unary representation of natural numbers and to plug a binary representation instead. However, at the moment, this requires the user to align the concepts of the library they have imported with the one in the target system. This may be a tiresome work. Another disadvantage is that the proofs we export are hardly maintainable because they are not idiomatic: Proof terms are exported for Coq, Lean and Matita. Generated tactic scripts is exported for PVS, and in OpenTheory, it is inherent to the system that proof script are not readable. More generally,

[1] www.logipedia.science

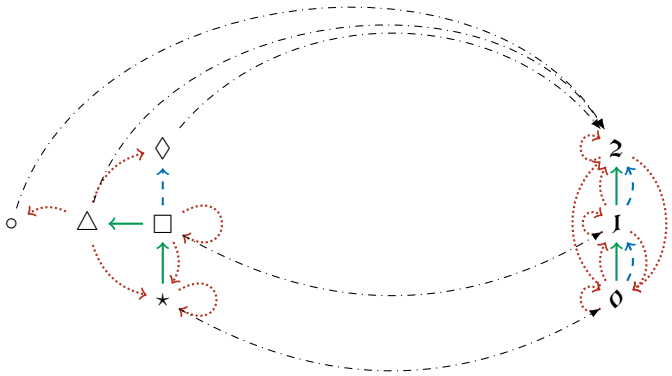


Figure 12.1: Sort-morphism from STT∀ to $\mathscr{C}_3$

```
Theorem congruent_exp_pred_S0 :
forall p a : nat, prime p -> Not (divides p a) ->
congruent (exp a (pred p)) (S 0) p.

theorem congruent_exp_pred_S0 :
forall (p:nat.nat) , forall (a:nat.nat) , primes.prime p ->
connectives.Not (primes.divides p a) ->
(cong.congruent (exp.exp a (nat.pred_ p)) (nat.S nat.O) p.

theorem congruent_exp_pred_S0 :
\forall (p:nat). \forall (a:nat).
prime p -> Not (divides p a) -> congruent (exp a (pred p)) (S 0)  p.
```

Figure 12.2: Coq, Lean and Matita output

our exportation does not genuinely use the high-level language of these systems offered to the user. At the end of this chapter, we discuss some ideas to overcome these problems.

12.1 From STT∀ to Coq, Lean and Matita

Taking the representation of STT∀ as a CTS, one can give a translation of STT∀ to the CTS $\mathscr{C}_3$ (1.5.12) as a sort-morphism as presented in Figure 12.1.

Since the CTS $\mathscr{C}_3$ is a subset of the CTS implemented by Coq, Lean, and Matita the translation to these systems is easy because a sort-morphism can always be applied on a proof term directly without need to type check it. With the encoding of STT∀ in Dedukti we have a direct representation of the proof term, we translate this proof term just as a string which, later, can be parsed by the target system. In Figure 12.2, we give the result of our translation applied to Fermat's little theorem into Coq, Lean and Matita.

Remark 29 *Apart from the syntax, the translation to Coq, Lean and Matita is the same except for one thing: In Lean, a parameter which returns a proposition needs to be prefixed by*

a keyword `noncomputable`

12.2 From STT∀ to PVS

The logic of PVS [ORS92] can be seen as a conservative extension of the PTS λHOL with predicate subtyping [Gil18] using a sequent calculus [GTL89]. Fortunately, since the version 7 of PVS, the system has the prenex polymorphism feature. Hence, the logic of STT∀ is strictly included into that of PVS.

The difficulty to export PVS proofs comes from the fact that the PVS system has been designed to interact with a human and not with a computer. In particular, the system has no proof term. Hence, the only way to use PVS is with tactics. However, even if the logic of PVS extends that of STT∀, there are no PVS tactics that strictly simulate the deduction rules of STT∀. PVS tactics tend to simplify the goal whenever it is possible. For example, a goal such as $A \wedge \top$ will be automatically simplified to A. Hence, translating STT∀ proofs in PVS by translating the deduction rules of STT∀ by a set of tactics often generates an erroneous PVS script.

Example 12.1 *As an example, assuming we have a proof π_1 in STT∀ of $\Gamma \vdash_{\mathscr{S}} A \to B \wedge \top$ and a proof π_2 of $\Gamma \vdash_{\mathscr{S}} A$. Then we can apply the rule $\mathscr{S}_{\Rightarrow_E}$ to deduce $\Gamma \vdash_{\mathscr{S}} B \wedge \top$. Because PVS uses sequent calculus, the $\mathscr{S}_{\Rightarrow_E}$ is translated with a cut rule as follows:*

$$\dfrac{\dfrac{\dfrac{|\pi_1|}{\Delta \vdash_{\mathscr{P}} A \to B \wedge \top}}{\Delta \vdash_{\mathscr{P}} A \to B \wedge \top, B \wedge \top}\ w\text{-}right \qquad \dfrac{\dfrac{\dfrac{|\pi_2|}{\Delta \vdash_{\mathscr{P}} A}}{\Delta \vdash_{\mathscr{P}} A, B \wedge \top}\ w\text{-}right \qquad \dfrac{}{\Delta, B \wedge \top \vdash_{\mathscr{P}} B \wedge \top}\ axiom}{\Delta, A \to B \wedge \top \vdash_{\mathscr{P}} B \wedge \top}\ \Rightarrow\text{-}left}{\Delta \vdash_{\mathscr{P}} B \wedge \top}\ cut$$

However, after the cut, PVS will automatically simplify the goal $B \wedge \top$ into B on the right premise. Therefore, we will not be able to apply the $\Rightarrow$-left rule as expected. An idea is to introduce a cut on A. Hence we replace the right derivation tree above by the following one:

$$\dfrac{\dfrac{\dfrac{\dfrac{|\pi_2|}{\Delta \vdash_{\mathscr{P}} A}}{\Delta, A \to B \wedge \top \vdash_{\mathscr{P}} A}\ w\text{-}left}{\Delta, A \to B \wedge \top \vdash_{\mathscr{P}} A, B \wedge \top}\ w\text{-}right \qquad \dfrac{}{\Delta, A \to B \wedge \top, A \vdash_{\mathscr{P}} B \wedge \top}\ \Rightarrow\text{-}left, axiom}{\Delta, A \to B \wedge \top \vdash_{\mathscr{P}} B \wedge \top}\ cut$$

However, we may see that introducing so many cuts (each time we use an elimination rule) is cumbersome and as such has a large impact on the type checking time as witnessed in Table 12.1.

12.3 From STT∀ to OpenTheory

The logic behind OpenTheory is similar to the one of STT∀ except for the three following differences:

- It uses only one connective: Equality.

$$\frac{}{t \vdash_{\mathscr{O}} t}\ \mathscr{O}_{assume}$$

$$\frac{\Gamma \vdash_{\mathscr{O}} t = u}{\Gamma \vdash_{\mathscr{O}} \lambda x.\, t = \lambda x.\, u}\ \mathscr{O}_{absThm}$$

$$\frac{\Gamma \vdash_{\mathscr{O}} t_1 = t_2 \qquad \Gamma \vdash_{\mathscr{O}} u_1 = u_2}{\Gamma \cup \Delta \vdash_{\mathscr{O}} t_1\ t_2 = u_1\ u_2}\ \mathscr{O}_{appThm}$$

$$\frac{}{t_1, \ldots, t_n \vdash_{\mathscr{O}} u}\ \mathscr{O}_{axiom}$$

$$\frac{}{\vdash_{\mathscr{O}} \lambda x.\, t\ u = t\,\{x \leftarrow u\}}\ \mathscr{O}_{beta}$$

$$\frac{\Gamma \vdash_{\mathscr{O}} t \qquad \Delta \vdash_{\mathscr{O}} u}{(\Gamma - u) \cup (\Delta - t) \vdash_{\mathscr{O}} t = u}\ \mathscr{O}_{deductAntiSym}$$

$$\frac{\Gamma \vdash_{\mathscr{O}} t \qquad \Delta \vdash_{\mathscr{O}} t = u}{\Gamma \cup \Delta \vdash_{\mathscr{O}} u}\ \mathscr{O}_{eqMp}$$

$$\frac{\Gamma \vdash_{\mathscr{O}} t \qquad \Delta \vdash_{\mathscr{O}} u}{(\Gamma - u) \cup \Delta \vdash_{\mathscr{O}} t}\ \mathscr{O}_{proveHyp}$$

$$\frac{}{\vdash_{\mathscr{O}} t = t}\ \mathscr{O}_{refl}$$

$$\frac{\Gamma \vdash_{\mathscr{O}} t}{\Gamma\sigma \vdash_{\mathscr{O}} t\sigma}\ \mathscr{O}_{subst}$$

$$\frac{\Gamma \vdash_{\mathscr{O}} t = u}{\Gamma \vdash_{\mathscr{O}} u = t}\ \mathscr{O}_{sym}$$

$$\frac{\Gamma \vdash_{\mathscr{O}} t_1 = t_2 \qquad \Gamma \vdash_{\mathscr{O}} t_2 = t_3}{\Gamma \vdash_{\mathscr{O}} t_1 = t_3}\ \mathscr{O}_{trans}$$

Figure 12.3: OpenTheory Typing system

- The logic is classical (it is based on the $\mathcal{Q}_0$ logic [And86]).
- There is no conversion in OpenTheory. β is axiomatized as long as the mechanism for global definitions (δ rewriting).

The logic of OpenTheory is presented in Figure 12.3.

12.3.1 Connectives of STT∀ into OpenTheory

We solve the first two points and the last point independently. For the first two points, one needs to find a sound encoding of the connectives of STT∀ in OpenTheory. STT∀ has three connectives: The implication $\Rightarrow$, the forall quantifier $\forall$ and the typed forall quantifier $\mathcal{V}$. It is known that implication and forall can be both encoded into $\mathcal{Q}_0$. For the last quantifier, this is also possible because in OpenTheory, $\mathcal{V}$ is not a connective since the quantification over type variables is implicit. The encoding is given below:

$$\begin{aligned}
\top &:= (=) = (=) \\
t \wedge u &:= \lambda f.\, f\ t\ u = \lambda f.\, f\ \top\ \top \\
t \Rightarrow u &:= t \wedge u = t \\
\forall x : A.\ u &:= \lambda x : A.\, u = \lambda x : A.\, \top \\
\mathcal{V} A.\ u &:= u
\end{aligned}$$

Notice that we have decided not to inline the definition of the connectives $\wedge$ and $\top$ for clarity. This encoding is sound because every elimination and introduction rules on these connectives are derivable in OpenTheory. We sketch the proofs here, all the details can be found in [Thi18].

◊ *Introduction of* $\wedge$:

	(1)	$\Gamma \vdash_{\mathscr{O}} t$	Main Hypothesis	
	(2)	$\Gamma \vdash_{\mathscr{O}} u$		
	(3)	$\Gamma \vdash_{\mathscr{O}} t = \top$	$\mathscr{O}_{eqMp}$	1
	(4)	$\Gamma \vdash_{\mathscr{O}} u = \top$	$\mathscr{O}_{eqMp}$	2
	(5)	$\Gamma \vdash_{\mathscr{O}} f = f$	$\mathscr{O}_{refl}$	
	(6)	$\Gamma \vdash_{\mathscr{O}} f\ t\ u = f\ \top\ \top$	$\mathscr{O}_{appThm}$	5,3,4
	(7)	$\Gamma \vdash_{\mathscr{O}} \lambda f.\, f\ t\ u = \lambda f.\, f\ \top\ \top$	$\mathscr{O}_{absThm}$	6
✯	(8)	$\Gamma \vdash_{\mathscr{O}} t \wedge u$	Definition of $\wedge$	7

◊ *Elimination of* $\wedge$:

Without loss of generality, we prove only the first projection.

	(1)	$\Gamma \vdash_{\mathscr{O}} t \wedge u$	Main Hypothesis	
	(2)	$\Gamma \vdash_{\mathscr{O}} (\lambda f.\, f\ t\ u)\,(\lambda x.\, \lambda y.\, x) = (\lambda f.\, f\ \top\ \top)\,(\lambda x.\, \lambda y.\, x)$	$\mathscr{O}_{appThm}$	1
	(3)	$\Gamma \vdash_{\mathscr{O}} t = \top$	$\mathscr{O}_{beta}$, $\mathscr{O}_{trans}$	2
✯	(4)	$\Gamma \vdash_{\mathscr{O}} t$	$\mathscr{O}_{eqMp}$	3

◊ *Introduction of* $\Rightarrow$ *(*$\mathscr{S}_{\Rightarrow_I}$*)*:

	(1)	$\Gamma, t \vdash_{\mathscr{O}} u$	Main hypothesis	
	(2)	$\Gamma, t \wedge u \vdash_{\mathscr{O}} t$	Elimination of $\wedge$	
	(3)	$\Gamma, t \vdash_{\mathscr{O}} t \wedge u$	Introduction $\wedge$	1
	(4)	$\Gamma \vdash_{\mathscr{O}} t \wedge u = t$	$\mathscr{O}_{deductAntiSym}$	2,3
✯	(5)	$\Gamma \vdash_{\mathscr{O}} t \Rightarrow u$	Definition of $\Rightarrow$	4

◊ *Elimination of* $\Rightarrow$ *(*$\mathscr{S}_{\Rightarrow_E}$*)*:

	(1)	$\Gamma \vdash_{\mathscr{O}} t \Rightarrow u$	Main hypothesis	
	(2)	$\Gamma \vdash_{\mathscr{O}} t$		
	(3)	$\Gamma \vdash_{\mathscr{O}} t \wedge u = t$	Definition of $\Rightarrow$	1
	(4)	$\Gamma \vdash_{\mathscr{O}} t \wedge u$	$\mathscr{O}_{eqMp}$	2,3
✯	(5)	$\Gamma \vdash_{\mathscr{O}} u$	Elimination of $\wedge$	4

◊ *Introduction of* $\forall$ *(*$\mathscr{S}_{\forall_I}$*)*:

	(1)	$\Gamma \vdash_{\mathscr{O}} t$	Main hypothesis	
	(2)	$\Gamma \vdash_{\mathscr{O}} t = \top$	$\mathscr{O}_{deductAntiSym}$	1
	(3)	$\Gamma \vdash_{\mathscr{O}} \lambda x.\, t = \lambda x.\, top$	$\mathscr{O}_{absThm}$	2
✯	(4)	$\Gamma \vdash_{\mathscr{O}} \forall x.\, t$	Definition of $\forall$	3

◊ *Elimination of* $\forall$ *(*$\mathscr{S}_{\forall_E}$*)*:

	(1)	$\Gamma \vdash_{\mathscr{O}} \forall x.\, t$	Main hypothesis	
	(2)	$\Gamma \vdash_{\mathscr{O}} u$		
	(3)	$\Gamma \vdash_{\mathscr{O}} \lambda x.\, t = \lambda x.\, top$	Definition of $\forall$	1
	(4)	$\Gamma \vdash_{\mathscr{O}} (\lambda x.\, t)\ u = (\lambda x.\, top)\ u$	$\mathscr{O}_{appThm}$	3
	(5)	$\Gamma \vdash_{\mathscr{O}} t\,\{x \leftarrow u\} = \top$	$\mathscr{O}_{beta}$, $\mathscr{O}_{trans}$	4
✯	(6)	$\Gamma \vdash_{\mathscr{O}} t\,\{x \leftarrow u\}$	$\mathscr{O}_{eqMp}$	5

12.3.2 Removing β and δ steps

Removing β and δ rewriting steps is a problem similar to the problem of equivalence between typed conversion and implicit conversion discussed in Chapter 3. However in this case, the problem is easier because there are no dependent types. Hence, the circularity we mentioned in Section 3.3 does not exist anymore. However, this is not a direct consequence of Vincent Siles' results [Sil10] because STT$\forall$ is not a PTS but a CTS. In STT$\forall$, the subtyping is only used to go from a monophormic type to a polymorphic type, but a β reduction or a δ reduction never changes the status of a type, hence a cast is never introduced through a reduction (but a cast may be duplicated). This means that in the case of STT$\forall$, the subject reduction property can be proven without introducing new conversions or new subtyping rules. Therefore, it becomes straightforward to see that the untyped conversion is equivalent to a typed conversion by induction. Details of this proof are given in [Thi18].

12.4 Concept alignment

Concept alignment is the problem of exporting proofs using concepts already defined in the target system or in its standard library.

If we look at the statement of Fermat's little theorem exported in Coq it looks like this:

```
Definition congruent_exp_pred_S0 : forall (p:nat.nat), forall (a:nat.nat),
↪  (primes.prime p) -> (connectives.Not (primes.divides p a)) ->
↪  cong.congruent (exp.exp a (nat.pred p)) (nat.S nat.O) p := ...
```

The constants `prime`, `congruent` and `pred` come with a definition while the constants `exp`, `Not`, `O` and `S` are axiomatized and should be defined by the user. This is because our export function does not see that `nat.nat` is an inductive type and that `exp` is a recursive function defined by two axioms. Hence, to use that theorem, the user needs to link these constants with the library it intends to use (for Coq, it might be the standard library of Coq, or the mathematical component library). For the arithmetic library we have exported, one has to define about 40 constants and prove about 80 axioms. We have made this instantiation with Coq standard library and it took us approximatively one hour. All the axioms can be proved via the reflexivity of the equality, except two which require an eta-expansion. Since the concepts used to prove Fermat's little theorem are fairly standard, it was easy to find their counterpart in the standard library of Coq. For example, the definition of `exp` has to satisfy the two following axioms:

```
Axiom sym_eq_exp_body_0 : forall n : nat, (S 0) = (exp n 0).
Axiom sym_eq_exp_body_S : forall n m : nat, (times (exp n m) n) = (exp n (S m)).
```

The definition of `exp` in the standard library of Coq is the following

```
Fixpoint exp (n m : nat) : Datatypes.nat :=
  match m with
  | 0 => S 0
  | S m => n * exp n m
  end
```

It is relatively easy to check that this recursive function satisfies the two axioms above and are even theorems in the standard library of Coq.

For larger libraries, the problem of alignment is an issue because this work needs to be made every time a user downloads the proof and there is currently no way to parameterize the export function to make this link automatic. We will come back to this issue in Section 12.7 where we propose some solutions to overcome this problem.

12.5 Logipedia: An Online Encyclopedia

Logipedia (www.logipedia.science) is a front-end to the transformations we have presented in this work, in particular Chapter 11 and this one. Using their browser, the user can search for a theorem and download a proof of a theorem into one of the following systems: Coq, Lean, Matita, OpenTheory and PVS. Besides making these proofs available on a website, we have used the web structure to represent the theorems and their dependencies contained in the proof of Fermat's little theorem. For each theorem, a web page contains the following information which will be detailed in the rest of this section:

- Its statement using an ad-hoc pretty printer.
- Its taxonomy.
- The *theory* in which this theorem is expressed.
- The *main dependencies* of a theorem.

Taxonomy: The taxonomy is an information related to an entry in Logipedia. The arithmetic library we have exported does not contain only theorems, but also definitions and axioms. The field *taxonomy* defines the *kinds* that an element may have. In STT$\forall$, the taxonomy defines 5 kinds of elements:

- A type operator (such as `nat`,`list`,`bool`).
- A parameter (such as `plus`).
- A definition (such as `2` which is defined as the successor of `1`).
- An axiom (such as $\forall x, 0 + x = x$).
- A theorem (such as Little Fermat's theorem).

Of course, every logic defines its own taxonomy. In Calculus of Inductive Constructions, there might be another taxonomy to take into account inductive types, recursive functions and more...

Theory: A theory is the context Γ in which the theorem is proven. This context tends to be big in STT$\forall$ because types and recursive functions are axiomatized. In the context of STT$\forall$ it is defined below.

Definition 12.5.1
Given a symbol A, its theory denoted by $Th(A)$ in Logipedia *is computed by the formula below:*

$$Th(A) := \begin{cases} \{D \mid \forall B, B \in Dep(A) \wedge D \in Th(B)\} \cup \{B\} & \text{if } B \text{ or } A \text{ are a parameter or an axiom} \\ \{D \mid \forall B, B \in Dep(A) \wedge D \in Th(B)\} & \text{otherwise} \end{cases}$$

where $Dep(A)$ is the set of direct dependencies of A.

The theory of an entry is defined recursively. However, we make a distinction on the kind of A. Indeed, if A is itself a parameter or an axiom, we need to take into account all the definitions (or even axioms for theories with dependent types) that may appear in their type. Hence, a theory may contain an axiom or a definition.

Main dependencies: A theorem may have many direct dependencies. However, for the website, we wanted to avoid printing all the dependencies. For example Fermat's little theorem uses the commutativity of addition. However, it is not relevant to print the commutativity of addition as a dependency here. This suggests defining a notion of *main dependencies* which roughly are dependencies that are not in the transitive closure of the other dependencies. It is formally defined below:

Definition 12.5.2
Given an entry A, its main dependencies denoted by $Md(A)$ in Logipedia *are computed by the formula below:*

$$Md(A) := \{B \mid B \in Dep(A) \wedge \forall C \in Md(A), C \neq B \Rightarrow B \notin Dep^*(C)\}$$

where $Dep^*(A)$ is the set of dependencies of A closed by transitivity and reflexivity.

However, this definition is not recursive and may not scale (computing the transitive closure of dependencies for every entry takes a lot of time). Instead, one may approximate this definition by looking at a bounded depth.

12.6 The website

The website itself has been written with the classical triptych languages: HTML/CSS, Javscript and PHP. Moreover, the current version of the website uses a database to store Logipedia entries and their dependencies. The database technology used is MongoDB but it could have been SQL. The reason behind the choice of MongoDB comes from a previous version of the website where proofs (as lambda-terms) were also stored in SQL and it seemed that MongoDB was better for that.

12.6.1 Storage of proofs

When the user clicks on the download button of a theorem, definition etc... They can download an archive which contains a proof of the theorem. The choice has been made that this archive contains only the necessary dependencies. These files can be generated in two ways:

	Dedukti[STT∀]	Higher-Order Logic	Coq	Matita	Lean	PVS
size (mb)	1.5	41	0.6	0.6	0.6	9
translation time (s)	-	18	3	3	3	3
checking time (s)	0.1	13	6	2	1	∼300

Table 12.1: Exportation time for Fermat's little theorem in STT∀

- When the user clicks on the download button, which triggers a function that generates these files.
- Or all the files that the user wants to download can be computed in advance.

A previous version of the website used the first method, however this requires storing proofs on the database which, in SQL, is a bad idea as discussed above. Moreover, generating these files can take a long time.

This is why we have chosen to generate all these files in advance for Fermat's little theorem. The time it takes to generate these files (from an STT∀ proof term encoded in Dedukti) is summed up in Table 12.1.

12.7 Future Work

Logipedia is a new project and can be extended in many ways. Here are some of them.

Exporting to HOL-Light: Our main interest behind having an export function to OpenTheory is that OpenTheory is interoperable with several systems of the HOL family: HOL-Light, Isabelle/HOL, HOL4 and Proof Power

Generalizing the Logipedia website from STT∀ to many theories: The first version of Logipedia was hard-coded with the STT∀ logic. However, this is not suitable in the long term because we would like to export proofs from any logic whenever it is possible. For example, STT∀ does not handle dependent types. Unplugging STT∀ is a work which has started already and the main difficulty is the web exportation. To maintain a proper web exportation we need two things:

- A pretty printer to print a readable theorem,
- Defining a taxonomy for the logic used by the proofs.

The pretty-printer allows the user to understand the statement of the theorem without knowing the syntax of Dedukti for example. The taxonomy allows classifying entries: Parameter, inductive types, constructors, theorems, ... In STT∀, `nat` is a type operator, `0` is a constant, `prime` is definable and `plus_n_0` is a theorem. This taxonomy may differ depending on the logic. In Calculus of Inductive Constructions for example, `nat` is an inductive type, and `0` is one of its constructors.

Of course, we would like to have a parametric website as well as a parametric translator to JSON files where neither the pretty printer nor the taxonomy is hard-coded. An idea would be to use meta rewriting with Dkmeta for example to define the taxonomy. In the case of STT∀ the taxonomy can be defined using two pieces of information: Is the current symbol defined? Does its type start by the constant `eps` or `etap`. In that case, in STT∀ the taxonomy would be:

```
[] taxonomy.get is_def    (sttfa.eps _) --> taxonomy.theorem.
[] taxonomy.get is_static (sttfa.eps _) --> taxonomy.axiom.
[] taxonomy.get is_def    (sttfa.eta _) --> taxonomy.definition.
[] taxonomy.get is_static (sttfa.eta _) --> taxonomy.constant.
[] taxonomy.get is_static (prod.prod _) --> taxonomy.type_operator.
```

The advantage of using Dkmeta here, is that is it easier to define a taxonomy via rewrite rules than forking the current Logipedia project and implementing the taxonomy. Moreover, the advantage is that the taxonomy can be changed easily without having to recompile the Logipedia project. It would be interesting to see whether this methodology can be extended to other logics than STT∀.

Uploading proofs to Logipedia Another question is how proofs could and should be added to Logipedia. Let us take an example. A user wishes to upload its Coq library on Logipedia. Fortunately, these proofs could also be expressed in STT∀. How and what/who should translate these proofs in STT∀? How can we handle new versions of the library? Should these proofs be considered as new proofs? This is also related to the question below about when two proofs should be considered equal.

An intermediate solution could be to tag with meta-data, indicating where it comes from and whether this proof has already been translated from another logic. But probably in the long term, this may require a version control system such as Git.

Equality between proofs There might be many proofs of the same theorem in Logipedia:

- A famous example is the Pythagorean theorem which has at least 112 proofs [Pow95].
- The same proof could be expressed in several logics. For example one may prove the Pythagorean theorem in ***ZFC***, but since the proof does not use the axiom of choice then it could also be proved in ***ZF***. Should these proofs be treated as two different proofs?
- Fermat's little theorem has two proofs in Logipedia: The one expressed in Matita and the one expressed in STT∀. Should both proofs be stored on Logipedia?

However, while in the first case it may seem interesting to have all these proofs in Logipedia, for the last two cases it is not clear because it seems that they are actually the *same* proof. And hence, this raises the question: When should two proofs in Logipedia be considered the same?

Considering that two proofs are equal if they are syntactically equal is not a reasonable definition. Indeed, it would mean that taking a proof and unfolding only one definition (a δ reduction) would lead to another proof.

Another idea is to consider that two proofs are equal when their normal form is the same. But this definition is restrictive:

- Computing the normal form is often too expensive,
- A proof could be expressed in a logic but its normal form could be expressed in a weaker logic.

This second point means that a proof using dependent types but no polymorphism and a proof using polymorphism but no dependent types could be considered as equal because their normal form is the same. But are they?

On the other hand, since a proof of STT∀ is always a proof in Coq, it seems redundant that Logipedia stores this proof twice.

In order to overcome this issue we could order logics according to the notion of logic extension ($\mathcal{L}_1 \subseteq \mathcal{L}_2$ when all the proofs of $\mathcal{L}_1$ are also a proof in $\mathcal{L}_2$), regardless of whether the extensions are conservative. Using this order, Logipedia should record a proof in a logic $\mathcal{L}$ only if it does not record a proof of the same statement in a logic $\mathcal{L}' \subseteq \mathcal{L}$. The notion behind *cannot be expressed* remains to be defined but roughly we could say that there is not sort-morphism from the free CTS generated by the proof term from Dedukti to the CTS behind $\mathcal{L}'$ for example.

Finally, another idea would be to use only δ reduction to compare two proofs and not β reductions. The reason is that β reductions may remove polymorphism from a proof or dependent types and hence, this may imply that the reduced proof could be expressed in a weaker logic. This is never the case with δ reductions.

Concept alignment Concept alignment (as mentioned in Section 12.4) is a big issue to make proofs in Logipedia usable. While we have shown that alignment was an issue concerning recursive functions and inductive types, there are other pieces of information which are not imported in Dedukti and therefore are missing in the exporation of Logipedia. For example:

- Notations,
- High-Level structures (type classes, functors),
- Implicit parameters.

Having an automatic translation which recovers these pieces of information seems really hard. We think that these pieces of information should be first imported via another translation while currently they are lost. This other translation could produce a new file using a new standard, a *structure file*[2]. Then, around the proofs transformations we should have another transformation to maintain these structures files. For example, when a definition is pruned with Dkprune, its informations related into the structure file should be pruned too; with Universo it seems that the structure file remained unchanged...

Such structure file could be used by the export function of Logipedia. The main advantage to keep these pieces of information in another file is that the user of Logipedia can choose the structure file he wants. In opposition to Dedukti files containing proofs, this file could be generated manually, could be partial, and could contain alignements. Once this file has been written, it can be reused by other people exporting proofs with Logipedia.

[2]This structure file may not use Dedukti's syntax.

Conclusion

Chapter 13

Conclusion

In this thesis, we have shown how interoperability between concrete systems at the type system level (in opposition to the user syntax level) could be achieved via the logical framework λΠ-calculus modulo theory. Interoperability was discussed both from a theoretic point of view and from a practical one.

In Part I, we showed that the logic behind many proof assistants that exist today could be seen as a Cumulative Type System with some extensions (such as inductive types, recursive functions or universe polymorphism). We showed how interoperability could be formulated for Cumulative Type Systems and gave an incomplete procedure that decides whether a proof from one Cumulative Type System could be encoded into another (Section 2.3). Then, we defined well-structured derivation trees (Definition 3.1.2). For well-structured derivation trees, we can derive an induction principle compatible with β reduction. We showed that this idea may lead to a new way to attack difficult conjectures such as expansion postponement (Theorem 3.2.4) or the equivalence between syntactic CTS using an untyped conversion with semantic CTS using a typed conversion (Theorem 3.3.8). We checked that the derivation trees we used in the second part of this work were well-structured. We also conjecture that any derivation trees are well-structured (Conjecture 9). Afterwards, we defined bi-directional CTS (Definition 4.1.1). Bi-directional CTS split the typing judgment of CTS into an inference judgment (without subtyping) and a checking judgment (with subtyping). We defined the class of CTS in normal form (Definition 4.2.1) for which there is an equivalence between bi-directional CTS and usual CTS(Theorem 4.3.9) if the derivation tree is well-structured. We showed that bi-directional CTS could be embedded into λΠ-calculus modulo theory (Definition 6.1.2). The soundness proof (Theorem 6.2.41) requires that the derivation tree is well-structured in the first place. Finally, we presented STT∀ (Section 7.1) as a new logic and we showed that it could be formulated as a CTS (Theorem 7.2.1).

In Part II, we described Dedukti, an implementation of λΠ-calculus modulo theory. We also proposed several tools to translate proofs inside the Dedukti framework. First of all, we proposed *higher-order rewriting* as a programming language to write proof transformations. This was implemented in a tool called Dkmeta. Dkmeta has other features such as a quoting/unquoting mechanism to circumvent limitations of Dedukti. We also described Universo, a Dedukti tool which implements an incomplete procedure which decides whether a CTS proof in Dedukti can be embedded into another CTS. This tool also works for logics that extends CTS with universe polymorphism, inductive types and recursive functions. Afterwards, we proposed a semi-automatic procedure to translate proofs from Matita to STT∀. We showed that this procedure is effective for arithmetic proofs, in particular a proof of Fermat's little theorem. Finally, we showed that proofs in STT∀ could be exported to different systems, namely: Coq,

Matita, Lean, OpenTheory and PVS. This led to a website, Logipedia where a user can download a proof that can be checked in one of these systems.

13.1 Future Work

This work could be extended in many ways:

- In chapter 2, we provided an incomplete procedure to decide whether a proof (as a judgment) expressed in the CTS $\mathscr{C}$ could be translated in the CTS $\mathscr{D}$. To recover completeness, we would need to solve conjecture 4 and conjecture 5 which imply the existence of a canonical derivation tree. The first conjecture states the existence of a partial order on derivation trees that forms a join-semilattice. The second conjecture states that this order is well-founded. We also left as an open problem whether the equivalences we showed could be used to adapt Barthe results [Bar99b] on the decidability of injective PTS and decides the type checking of a large class of CTS.

- In chapter 3, we defined well-structured derivation trees , a predicate over derivation trees which attach a level to a derivation tree and consequently to a judgment. Roughly, a derivation tree is well-structured if the ordering generated by the *has type* relation $\prec$ (Definition 3.1.1) is well-founded and compatible with β. We showed in this chapter, that any well-structured derivation tree also satisfies the expansion postponement conjecture and could be translated to a CTS with a typed conversion. We also provided an empirical evidence that many proofs used in practice are well-structured. We conjecture that any derivation tree is actually well-structured. If this conjecture is true, it would provide a new induction scheme for derivation trees. Intuitively, levels memorize the complexity of the terms and types which appear in a deriation tree. If a judgment has a well-structured derivation tree, then in particular it admits a minimal level. In that case, it would be interesting to investigate the meaning of this minimal level.

- In Chapter 4, we introduced bi-directional CTS which refine the typing judgment of CTS with two new judgments: An *inference* judgment and a *checking* judgment. In this type system, subtyping can be used only during an application or at the end of a derivation. We defined a large class of CTS (namely *normal* CTS) for which there is an equivalence between bi-directional CTS and CTS as presented in Chapter 2. We conjectured that any CTS specification is weakly equivalent to a CTS in normal form. This conjecture could perhaps be solved using results we introduced in Chapter 4. Also, it would be interesting to understand whether this equivalence is true for a larger class of CTS.

- In Chapter 5, we introduced PTS modulo as an extension of PTS with an abstract conversion. This system is a reformulation of the one introduced by Frédéric Blanqui in [Bla01]. In our system, equations are added one by one into the context, this presentation is closer to concrete implementations such as Dedukti. Then, we introduced $\lambda\Pi$-calculus modulo theory as an instance of a PTS modulo which corresponds to LF. Then, we introduced *shallow encodings* (Definition 5.2.1) which translate a judgment into a judgment. Using Cousineau & Dowek results [CD07], PTS can be encoded in a shallow way to $\lambda\Pi$-calculus modulo theory. However, the meta-theory of $\lambda\Pi$-calculus modulo theory relies on the injectivity of product which in general is hard to prove for concrete encodings. It would be interesting to investigate whether the notion of well-structured derivation trees introduced in Chapter 3 could help proving this property.

- In Chapter 6, we gave a parametric (with respect to the specification) encoding of CTS into λΠ-CALCULUS MODULO THEORY. Then, we gave a soundness proof for this encoding assuming that the input derivation trees are well-structured. We conjectured that this proof could also be reformulated to use the equivalence between semantic CTS and syntactic CTS instead. We also conjectured the completeness of this encoding and it would be interesting to see whether Ali Assaf's completeness proof for PTS could be extended to this encoding. Finally, the encoding functions are partial functions over the judgments of bi-directional CTS. One could express this translation function over CTS derivation trees directly. This way, we would avoid one indirection with bi-directional CTS and obtain a direct translation into λΠ-CALCULUS MODULO THEORY. However, this adds some complexity in the soundness proof because it requires manipulating function over derivation trees.

- In Chapter 7, we formalized STT∀, an extented version of SIMPLE TYPE THEORY with prenex polymorphism and type constructors. This logic aims to be a constructive version of HIGHER-ORDER LOGIC. We showed that this logic could also be expressed as a CTS and also as an extension of the λHOL PTS. Therefore, one can use results of Chapter 6 to embed this logic into λΠ-CALCULUS MODULO THEORY. The CTS view of STT∀ enables us to see possible extensions of polymorphism in STT∀ by allowing polymorphism of rank n and higher-order type variables. We showed that the first presentation of STT∀ can be represented as a CTS specification. We proved the soundness of this encoding and left as a conjecture its completeness. We argued in Chapter 9 that the completeness is true but a paper proof is missing. Moreover, it would be interesting to show whether the extensions we introduced of STT∀ using the CTS presentations are conservative.

- In Chapter 8, we presented DEDUKTI as an implementation of λΠ-CALCULUS MODULO THEORY.

- In Chapter 9, we presented a tool for DEDUKTI called DKMETA. DKMETA is built around the kernel of DEDUKTI and uses its rewrite engine extensively. We also extended DKMETA with a quote/unquote mechanism to get around limitations of the rewrite engine of DEDUKTI. This allows rewriting a syntactic application for example. DKMETA has many applications but we saw in Chapter 11 that it has some limitations. Here are three possible extensions of the tool:

  - Extend DKMETA so that we can also rewrite top-levels commands of DEDUKTI.
  - The quote/unquote mechanism of DKMETA is currently hard-coded in DKMETA. It would be interesting to have a feature that enables the user, in DKMETA, to define its own quote/unquote mechanism.
  - DKMETA is a meta language for DEDUKTI. As such, we think it could be used also as a refiner (a tool which fills holes left by the user). For example it would be interesting to implement meta-variables for DEDUKTI using DKMETA.

- In Chapter 10, we introduced UNIVERSO, another tool for DEDUKTI that aims at translating proofs from one CTS to another. This tool implements the procedure introduced in Chapter 2 and can be seen as an extension of COQ algorithm to check consistency with floating universes. In Chapter 10, we already discussed several ways to enhance the tool, but the main challenge that may be raised by big libraries is scalability. We observe today that the main roadblock of UNIVERSO is the time it takes for an SMT solver to solve the constraints. However, there are several ways to handle this problem:

  - Even if in general, universes are not modular, in practice, as proofs are split into libraries, we think that there are ways to manage the scalability by giving smaller problems to the SMT solver.
  - The Z3 solver has been used by F* to verify large programs. Hence, this proves that Z3 may scale to large problems. It would be interesting to investigate whether the solver could be tuned specifically to our CTS problems.
  - Universo can use Dkmeta to pre-process and post-process the entries. It would be interesting to see whether there are smart ways to pre-process the entries to reduce solving time.

- In Chapter 11, we presented a semi-automatic translation from the embedding of Matita to the embedding of STT∀ in Dedukti. This semi-automatic translation requires several passes with tools such as Dkprune, Dkmeta and Universo. While there are arguments on paper to show the that these tools can be reused for other translations and encodings, the tools we presented only have been tested mostly on the encoding of Matita except for two cases:

  - Some experiments have been made using Universo with proofs coming from Coq.
  - Dkmeta was used to translate proofs coming from the embedding of Higher-Order Logic [AB15] to the embedding of STT∀.

  Currently, the translation presented in this chapter can only be parameterized by someone who knows Dedukti well and the various embeddings involved in the translation. It would be interesting to see whether a nice user interface could be provided to parameterize these translations. For example via the Logipedia project.

- In Chapter 12, we presented Logipedia, an online encyclopedia of proofs that could be shared between several proof systems. One feature is that exported proofs are not obfuscated by the encodings we used through Dedukti. However, the proofs we exported are not ready to use as a library because we lost the structure of the library. For example, we do not provide any concept alignement:

  - In Dedukti, inductive types and recursive functions are axiomatized. For example, natural numbers come with a type declaration, two declarations for the constructors, and one axiom for the inductive principle. Moreover each recursive function such as *plus* is defined with two axioms (one for each constructor). However, once the proofs are exported, it is relatively easy to align these concepts with thos of the system.
  - Structural information such as implicit arguments, notations are lost through the translations and therefore are not exported.

  As we mentioned in the introduction, these problems arise because we only translate in Dedukti proofs intelligible by the type system supporting the proof systems. Hence the proofs we encode in Dedukti are not the ones written by the user. This is why the exported proofs look rather different than those that a user could write manually. One way to recover these pieces of structural information would be to record them as meta-data, and have a second translation on top of the existing one, translating the meta-data in the source system into meta-data in the target system. Hence, the exportation would know whether a Dedukti declaration should be interpreted as an inductive type or not for example. Moreover, we believe that these meta-data could be written manually for the exportation only with Logipedia.

13.2 Future of interoperability

In computer science, and especially in software engineering, standards are the rule: In programming language (C++ standards [ISO18], Javascript [ECM11]), in networks (RFCs [EF14], TCP/IP [Fei00]), or in operating systems (POSIX [Ins93]) to name a few. It is impressive that these standards are interoperable:

- Two different machines with two completely different systems may exchange data through a network as internet.
- A program written in C may communicate with a programming written in OCaml. Either via the file system or directly at compile time to produce only one binary executable. And this is true for many if not all programming languages.
- One may translate a document written in markdown into a LaTeX document or an HTML webpage via Pandoc [Dom14].

The world of formal proofs appears as an exception to this empirical fact. At the time of writing, almost no standard exist (except the code of the software) and interoperability between these systems has the reputation to be an impossible problem. One example is the `QED manifesto` project which started in 1993 and stopped in 1996 without tangible results.

In the last 20 years, many formal systems saw the light and we can observe empirically that the libraries built on top of these systems were very similar. In this work, we gave an explanation to this emprical fact: The logic used to build these libraries are similar. In this thesis we pointed out that the CTS framework is a good basis to study these proofs. But not only, the CTS framework also has the advantage of highlighting the differences between formal systems. The fact that we were able to partially automatize the translation of Fermat's little theorem and exporting this proof towards 5 different formal proof assistants shed a new light on interoperability at the proof level.

This experiment is a first step before conducting the interoperability at a larger scale to tackle big libraries of formal proofs such as the Mathematical Component library [MT17], The Mizar library [Rud92] or the Archive of Formal Proofs [LL10].

In the world of formal proofs, interoperability at proof level is already a big challenge but is not the only one. In particular, the proofs we exported contain too many details for a human and needs to be aligned with the concepts in the target system. In practice, libraries of formal proofs are built using a high-level language which is different from the type system used by the logic. Depending on the formal prover, these high-level languages differ completely. In Hol-light [Har09], this meta language is OCaml. But in Coq [BGG+14], they combine both a vernacular and a tactic language. In Agda [Nor09], this high-level language is provided via a vernacular and the interaction with the text editor. These high-level languages are essential to give some structure to the library but also to omit pieces of informations that can be reconstructed by the formal system. To have a real and practical interoperability between formal systems, this challenge is the last big piece missing in the puzzle of interoperability between proof systems.

The best way to predict the future is to implement it.

David Heinemeier Hansson

Bibliography

[AB15] Ali Assaf and Guillaume Burel. Translating HOL to dedukti. In Cezary Kaliszyk and Andrei Paskevich, editors, *Proceedings Fourth Workshop on Proof eXchange for Theorem Proving, PxTP 2015, Berlin, Germany, August 2-3, 2015*, volume 186 of *EPTCS*, pages 74–88, 2015.

[ABC+16] A. Assaf, G. Burel, R. Cauderlier, D. Delahaye, G. Dowek, C. Dubois, F. Gilbert, P. Halmagrand, O. Hermant, and R. Saillard. Dedukti: a logical framework based on the lambda-Pi-calculus modulo theory. Manuscript, 2016.

[ABC+18] Abhishek Anand, Simon Boulier, Cyril Cohen, Matthieu Sozeau, and Nicolas Tabareau. Towards certified meta-programming with typed template-coq. In Avigad and Mahboubi [AM18], pages 20–39.

[AC07] Andreas Abel and Thierry Coquand. Untyped algorithmic equality for martin-löf's logical framework with surjective pairs. *Fundam. Inform.*, 77(4):345–395, 2007.

[ACP16] *ACPH: System description for CoCo 2016*, 2016.

[ADJL16] Ali Assaf, Gilles Dowek, Jean-Pierre Jouannaud, and Jiaxiang Liu. Untyped Confluence in Dependent Type Theories. In *Proceedings Higher-Order Rewriting Workshop*, Proc. Higher-Order rewriting Workshop, Porto, Portugal, June 2016. EasyChair.

[AF18] June Andronick and Amy P. Felty, editors. *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*. ACM, 2018.

[AGM92] S. Abramsky, Dov M. Gabbay, and S. E. Maibaum, editors. *Handbook of Logic in Computer Science (Vol. 2)*. Oxford University Press, Inc., New York, NY, USA, 1992.

[Agu16] Alejandro Aguirre. Towards a provably correct encoding from F* to SMT. Inria Internship Report, August 2016.

[Alt15] Thorsten Altenkirch, editor. *13th International Conference on Typed Lambda Calculi and Applications, TLCA 2015, July 1-3, 2015, Warsaw, Poland*, volume 38 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[AM18] Jeremy Avigad and Assia Mahboubi, editors. *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10895 of *Lecture Notes in Computer Science*. Springer, 2018.

[And86] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth through Proof*. Academic Press Professional, Inc., USA, 1986.

[ARCT09] Andrea Asperti, Wilmer Ricciotti, C Sacerdoti Coen, and Enrico Tassi. A compact kernel for the calculus of inductive constructions. *Sadhana*, 34(1):71–144, 2009.

[ASCTZ07] Andrea Asperti, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. User interaction with the matita proof assistant. *J. Autom. Reason.*, 39(2):109–139, August 2007.

[Ass14] Ali Assaf. A calculus of constructions with explicit subtyping. In *TYPES*, volume 39 of *LIPIcs*, pages 27–46. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2014.

[Ass15a] Ali Assaf. Conservativity of embeddings in the lambda pi calculus modulo rewriting. In Altenkirch [Alt15], pages 31–44.

[Ass15b] Ali Assaf. *A framework for defining computational higher-order logics. (Un cadre de définition de logiques calculatoires d'ordre supérieur)*. PhD thesis, École Polytechnique, Palaiseau, France, 2015.

[Bar96a] B. Barras. Coq en coq. Rapport de Recherche 3026, INRIA, October 1996.

[Bar96b] B. Barras. Verification of the interface of a small proof system in coq. In E. Gimenez and C. Paulin-Mohring, editors, *Proceedings of the 1996 Workshop on Types for Proofs and Programs*, pages 28–45, Aussois, France, December 1996. Springer-Verlag LNCS 1512.

[Bar99a] B. Barras. *Auto-validation d'un système de preuves avec familles inductives*. Thèse de doctorat, Université Paris 7, November 1999.

[Bar99b] Gilles Barthe. Type-checking injective pure type systems. *J. Funct. Program.*, 9(6):685–698, 1999.

[BB12] Mathieu Boespflug and Guillaume Burel. Coqine: Translating the calculus of inductive constructions into the λΠ-calculus modulo. *CEUR Workshop Proceedings*, 878, 06 2012.

[BBFM99] Patrick Behm, Paul Benoit, Alain Faivre, and Jean-Marc Meynadier. Meteor: A successful application of b in a large project. In *International Symposium on Formal Methods*, pages 369–387. Springer, 1999.

[BDL08] Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe, editors. *Logic and Theory of Algorithms, 4th Conference on Computability in Europe, CiE 2008, Athens, Greece, June 15-20, 2008, Proceedings*, volume 5028 of *Lecture Notes in Computer Science*. Springer, 2008.

[BGG+14] Pierre Boutillier, Stephane Glondu, Benjamin Grégoire, Hugo Herbelin, Pierre Letouzey, Pierre-Marie Pédrot, Yann Régis-Gianas, Matthieu Sozeau, Arnaud Spiwack, and Enrico Tassi. Coq 8.4 Reference Manual. Research report, Inria, July 2014. The Coq Development Team.

[BGH19] Frédéric Blanqui, Guillaume Genestier, and Olivier Hermant. Dependency pairs termination in dependent type theory modulo rewriting. In Geuvers [Geu19], pages 9:1–9:21.

[BHHJ11] Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors. *Third International Symposium on NASA Formal Methods (NFM 2011)*, volume 6617 of *Lecture Notes in Computer Science*. Springer, April 2011.

[BHKS19] Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen, editors. *Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III*, volume 11429 of *Lecture Notes in Computer Science*. Springer, 2019.

[BHS01a] Gilles Barthe, John Hatcliff, and Morten Heine Sørensen. Weak normalization implies strong normalization in a class of non-dependent pure type systems. *Theoretical Computer Science*, 269(1):317 – 361, 2001.

[BHS01b] Gilles Barthe, John Hatcliff, and Morten Heine B. Sørensen. An induction principle for pure type systems. *Theoretical Computer Science*, 266(1):773 – 818, 2001.

[BKV10] Matthias Blume, Naoki Kobayashi, and Germán Vidal, editors. *Functional and Logic Programming, 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19-21, 2010. Proceedings*, volume 6009 of *Lecture Notes in Computer Science*. Springer, 2010.

[Bla01] Frédéric Blanqui. *Théorie des types et réécriture. (Type theory and rewriting)*. PhD thesis, University of Paris-Sud, Orsay, France, 2001.

[BN99] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge university press, 1999.

[BRA13] EDWIN BRADY. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, 9 2013.

[BST10] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010.

[Can74] *On a Property of the Class of all Real Algebraic Numbers.*, 1874.

[Cau16a] Raphaël Cauderlier. *Object-Oriented Mechanisms for Interoperability between Proof Systems*. Theses, Conservatoire National Des Arts et Métiers, Paris, October 2016.

[Cau16b] Raphaël Cauderlier. A rewrite system for proof constructivization. In Dowek et al. [DLA16], pages 2:1–2:7.

[Cau18] Raphaël Cauderlier. Tactics and certificates in meta dedukti. In Avigad and Mahboubi [AM18], pages 142–159.

[CD07] Denis Cousineau and Gilles Dowek. Embedding pure type systems in the lambda-pi-calculus modulo. In *TLCA*, volume 4583 of *Lecture Notes in Computer Science*, pages 102–117. Springer, 2007.

[CDMM10] James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. The gentle art of levitation. In *ACM Sigplan Notices*, volume 45, pages 3–14. ACM, 2010.

[CH86] Thierry Coquand and Gérard Huet. *The calculus of constructions*. PhD thesis, INRIA, 1986.

[CH94] Thierry Coquand and Hugo Herbelin. A - translation and looping combinators in pure type systems. *J. Funct. Program.*, 4(1):77–88, 1994.

[Che98] Gang Chen. Dependent type system with subtyping (i) type level transitivity elimination. *Journal of Computer Science and Technology*, 13(6):564, 1998.

[Chu36] Alonzo Church. An unsolvable problem of elementary number theory. *American journal of mathematics*, 58(2):345–363, 1936.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *The journal of symbolic logic*, 5(2):56–68, 1940.

[Coq86] T. Coquand. An analysis of Girard's paradox. Technical Report RR-0531, INRIA, May 1986.

[Cur34] Haskell B Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences of the United States of America*, 20(11):584, 1934.

[DB72] Nicolaas Govert De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. Elsevier, 1972.

[DHK01] Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Hol-$\lambda\sigma$: an intentional first-order expression of higher-order logic. *Mathematical Structures in Computer Science*, 11(1):21–45, 2001.

[DLA16] Gilles Dowek, Daniel R. Licata, and Sandra Alves, editors. *Proceedings of the Eleventh Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, LFMTP 2016, Porto, Portugal, June 23, 2016*. ACM, 2016.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[dMKA+15] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In *International Conference on Automated Deduction*, pages 378–388. Springer, 2015.

[DNS95] Peter Dybjer, Bengt Nordström, and Jan M. Smith, editors. *Types for Proofs and Programs, International Workshop TYPES'94, Båstad, Sweden, June 6-10, 1994, Selected Papers*, volume 996 of *Lecture Notes in Computer Science*. Springer, 1995.

[Dom14] Massimiliano Dominici. An overview of pandoc. *TUGboat*, 35(1):44–50, 2014.

[ECM11] ECMA International. *Standard ECMA-262 - ECMAScript Language Specification*. 5.1 edition, June 2011.

[EF14] RFC Editor and Heather Flanagan. RFC Style Guide. RFC 7322, September 2014.

[Far08] William M. Farmer. The seven virtues of simple type theory. *Journal of Applied Logic*, 6(3):267 – 286, 2008.

[Fei00] Sidnie Feit. *TCP/IP*. McGraw-Hill, Inc., USA, 2000.

[FGH+88] Amy P. Felty, Elsa L. Gunter, John Hannan, Dale Miller, Gopalan Nadathur, and Andre Scedrov. Lambda-prolog: An extended logic programming language. In Lusk and Overbeek [LO88], pages 754–755.

[FJ19] Gaspard Férey and Jean-Pierre Jouannaud. Confluence in untyped higher-order theories: Part i. *ACM Transactions on Computational Logic*, 2019.

[Fre93] Gottlob Frege. *Grundgesetze der Arithmetik*, volume I. Verlag Hermann Pohle, Jena, 1893.

[FS18] Denis Firsov and Aaron Stump. Generic derivation of induction for impredicative encodings in cedille. In Andronick and Felty [AF18], pages 215–227.

[GAA+13] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, et al. A machine-checked proof of the odd order theorem. In *International Conference on Interactive Theorem Proving*, pages 163–179. Springer, 2013.

[Geu92] Herman Geuvers. The church-rosser property for beta-eta-reduction in typed lambda-calculi. In LICS 1992 [LIC92], pages 453–460.

[Geu93] Jan Herman Geuvers. *Logics and type systems*. [Sl: sn], 1993.

[Geu94] Herman Geuvers. A short and flexible proof of strong normalization for the calculus of constructions. In Dybjer et al. [DNS95], pages 14–38.

[Geu09] Herman Geuvers. Proof assistants: History, ideas and future. *Sadhana*, 34(1):3–25, 2009.

[Geu19] Herman Geuvers, editor. *4th International Conference on Formal Structures for Computation and Deduction, FSCD 2019, June 24-30, 2019, Dortmund, Germany*, volume 131 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019.

[Gil18] Frédéric Gilbert. *Extending higher-order logic with predicate subtyping: Application to PVS. (Extension de la logique d'ordre supérieur avec le sous-typage par prédicats)*. PhD thesis, Sorbonne Paris Cité, France, 2018.

[Gim94] Eduarde Giménez. Codifying guarded definitions with recursive schemes. In *International Workshop on Types for Proofs and Programs*, pages 39–59. Springer, 1994.

[Gir72] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, PhD thesis, Université Paris VII, 1972.

[GJCP19] Armaël Guéneau, Jacques-Henri Jourdan, Arthur Charguéraud, and François Pottier. Formal proof and analysis of an incremental cycle detection algorithm. In Harrison et al. [HOT19], pages 18:1–18:20.

[GN91] Herman Geuvers and Mark-Jan Nederhof. Modular proof of strong normalization for the calculus of constructions. *Journal of Functional Programming*, 1(2):155â€“189, 1991.

[God25] *Journal für die reine und angewandte Mathematik*. de Gruyter, 1925.

[God28] *Mathematische Zeitschrift*, Berlin, 1928.

[God92] Kurt Godel. *On Formally Undecidable Propositions of Principia Mathematica and Related Systems*. Dover Publications, 1992. Translation B. Meltzer.

[Gog95a] Healfdene Goguen. The metatheory of utt. In Peter Dybjer, Bengt Nordström, and Jan Smith, editors, *Types for Proofs and Programs*, pages 60–82, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[Gog95b] Healfdene Goguen. Typed operational semantics. In Mariangiola Dezani-Ciancaglini and Gordon Plotkin, editors, *Typed Lambda Calculi and Applications*, pages 186–200, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[Gon] Georges Gonthier. A computer-checked proof of the four colour theorem.

[GTL89] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and types*, volume 7. Cambridge University Press, 1989.

[Hal05] Thomas C Hales. A proof of the kepler conjecture. *Annals of mathematics*, 162(3):1065–1185, 2005.

[Hal17] Paul R Halmos. *Naive set theory*. Courier Dover Publications, 2017.

[Har09] John Harrison. Hol light: An overview. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 60–66, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[Har16] Robert Harper. *Practical foundations for programming languages*. Cambridge University Press, 2016.

[HHP93a] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM (JACM)*, 40(1):143–184, 1993.

[HHP93b] Robert Harper, Furio Honsell, and Gordon D. Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, 1993.

[Hon19] Gabriel Hondet. Efficient rewriting using decision trees. Master's thesis, ENAC ; Univesité Toulouse III- Paul Sabatier, August 2019.

[HOT19] John Harrison, John O'Leary, and Andrew Tolmach, editors. *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA*, volume 141 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[How80] *The formulae-as-types notion of construction*, 1980.

[Hur95] Antonius J. C. Hurkens. A simplification of girard's paradox. In Mariangiola Dezani-Ciancaglini and Gordon Plotkin, editors, *Typed Lambda Calculi and Applications*, pages 266–278, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[Hur11] Joe Hurd. The OpenTheory standard theory library. In Bobaru et al. [BHHJ11], pages 177–191.

[Ins93] American National Standards Institute. *IEEE standard for information technology: Portable Operating System Interface (POSIX) : part 2, shell and utilities*. IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, September 1993. Two volumes. IEEE Std 1003.2-1992 (includes IEEE Std 1003.2a-1992). Approved September 17, 1992, IEEE Standards Board. Approved April 5, 1993, American National Standards Institute. The primary purpose of this standard is to define a standard interface and environment for application programs that require the services of a 'shell' command language interpreter and a set of common utility programs.

[Irv95] *Russell's paradox*, 1995.

[ISO18] ISO. *ISO/IEC 9899:2018 Information technology — Programming languages — C*. Fourth edition, June 2018.

[Jec13] Thomas Jech. *Set theory*. Springer Science & Business Media, 2013.

[Jut93] L.S.V. Jutting. Typing in pure type systems. *Information and Computation*, 105(1):30 – 41, 1993.

[KdV89] Jan Willem Klop and Roel C. de Vrijer. Unique normal forms for lambda calculus with surjective pairing. *Inf. Comput.*, 80(2):97–113, 1989.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220, 2009.

[Kir18] Hélène Kirchner, editor. *3rd International Conference on Formal Structures for Computation and Deduction, FSCD 2018, July 9-12, 2018, Oxford, UK*, volume 108 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.

[KMNO14] Ramana Kumar, Magnus O Myreen, Michael Norrish, and Scott Owens. Cakeml: a verified implementation of ml. *ACM SIGPLAN Notices*, 49(1):179–191, 2014.

[LA04] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[Las12] Marc Lasson. *Réalisabilité et paramétricité dans les systèmes de types purs. (Realizability and parametricity in Pure Type Systems)*. PhD thesis, École normale supérieure de Lyon, France, 2012.

[LDF+18] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The OCaml system release 4.07: Documentation and user's manual. Intern report, Inria, July 2018.

[Len06] Stéphane JE Lengrand. *Normalisation & Equivalence in Proof Theory & Type Theory*. PhD thesis, University of St Andrews, 2006.

[Ler16] *CompCert-a formally verified optimizing compiler*, 2016.

[Let08] Pierre Letouzey. Extraction in coq: An overview. In Beckmann et al. [BDL08], pages 359–369.

[LIC92] *Proceedings of the Seventh Annual Symposium on Logic in Computer Science (LICS '92), Santa Cruz, California, USA, June 22-25, 1992*. IEEE Computer Society, 1992.

[LIC19] *34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, BC, Canada, June 24-27, 2019*. IEEE, 2019.

[LL10] Peter Lammich and Andreas Lochbihler. The isabelle collections framework. In *International Conference on Interactive Theorem Proving*, pages 339–354. Springer, 2010.

[LO88] Ewing L. Lusk and Ross A. Overbeek, editors. *9th International Conference on Automated Deduction, Argonne, Illinois, USA, May 23-26, 1988, Proceedings*, volume 310 of *Lecture Notes in Computer Science*. Springer, 1988.

[LR18] Rodolphe Lepigre and Christophe Raffalli. Abstract representation of binders in ocaml using the bindlib library. In Frédéric Blanqui and Giselle Reis, editors, *Proceedings of the 13th International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, LFMTP@FSCD 2018, Oxford, UK, 7th July 2018.*, volume 274 of *EPTCS*, pages 42–56, 2018.

[Mar08] Luc Maranget. Compiling pattern matching to good decision trees. In *Proceedings of the 2008 ACM SIGPLAN workshop on ML*, pages 35–46, 2008.

[MLS84] Per Martin-Löf and Giovanni Sambin. *Intuitionistic type theory*, volume 9. Bibliopolis Naples, 1984.

[MNS19] Aart Middeldorp, Julian Nagele, and Kiraku Shintani. Confluence competition 2019. In Beyer et al. [BHKS19], pages 25–40.

[MP00] Ieke Moerdijk and Erik Palmgren. Wellfounded trees in categories. *Annals of Pure and Applied Logic*, 104(1-3):189–218, 2000.

[MR86] Albert R. Meyer and Mark B. Reinhold. "type" is not a type. In *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '86, pages 287–295, New York, NY, USA, 1986. ACM.

[MT17] Assia Mahboubi and Enrico Tassi. Mathematical components, 2017.

[NFM17] Julian Nagele, Bertram Felgenhauer, and Aart Middeldorp. CSI: New evidence — a progress report. In Leonardo de Moura, editor, *Proceedings of the 26th International Conference on Automated Deduction (CADE-26)*, volume 10395 of *Lecture Notes in Artificial Intelligence*, pages 385–397, 2017.

[Nor09] Ulf Norell. Dependently typed programming in agda. In *Proceedings of the 6th International Conference on Advanced Functional Programming*, AFP'08, pages 230–266, Berlin, Heidelberg, 2009. Springer-Verlag.

[NPW02] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[ORS92] Sam Owre, John M Rushby, and Natarajan Shankar. Pvs: A prototype verification system. In *International Conference on Automated Deduction*, pages 748–752. Springer, 1992.

[ORSVH95] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich Von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of pvs. *IEEE Transactions on Software Engineering*, 21(2):107–125, 1995.

[Pea89] G. Peano. *Arithmetices principia: nova methodo exposita*. Nineteenth Century Collections Online (NCCO): Science, Technology, and Medicine: 1780-1925. Fratres Bocca, 1889.

[Pie10] Brigitte Pientka. Beluga: Programming with dependent types, contextual data, and contexts. In Blume et al. [BKV10], pages 1–12.

[PM96] C. Paulin-Mohring. *Définitions Inductives en Théorie des Types d'Ordre Supérieur*. Habilitation à diriger les recherches, Université Claude Bernard Lyon I, December 1996.

[Pol92] Randy Pollack. Typechecking in pure type systems. In *WORKSHOP ON*, page 271, 1992.

[Pol98] Erik Poll. Expansion postponement for normalising pure type systems. *Journal of Functional Programming*, 8(1):89–96, 1998.

[Pow95] Anthony Powell. Pythagorean theorem, 1995.

[PT00] Benjamin C Pierce and David N Turner. Local type inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1):1–44, 2000.

[PTA+19] Brigitte Pientka, David Thibodeau, Andreas Abel, Francisco Ferreira, and Rébecca Zucchini. A type theory for defining logics and proofs. In LICS 2019 [LIC19], pages 1–13.

[Rey74] John C Reynolds. Towards a theory of type structure. In *Programming Symposium*, pages 408–425. Springer, 1974.

[Ros39] J. Barkley Rosser. An informal exposition of proofs of godel's theorems and church's theorem. *J. Symb. Log.*, 4(2):53–60, 1939.

[Rud92] Piotr Rudnicki. An overview of the mizar project. In *Proceedings of the 1992 Workshop on Types for Proofs and Programs*, pages 311–330. Citeseer, 1992.

[Sai15] Ronan Saillard. *Typechecking in the lambda-Pi-Calculus Modulo : Theory and Practice. (Vérification de typage pour le lambda-Pi-Calcul Modulo : théorie et pratique)*. PhD thesis, Mines ParisTech, France, 2015.

[SH10] Vincent Siles and Hugo Herbelin. Equality is typable in semi-full pure type systems. In *Proceedings, 25th Annual IEEE Symposium on Logic in Computer Science (LICS '10), Edinburgh, UK, 11-14 July 2010*. IEEE Computer Society Press, 2010.

[Sil10] Vincent Siles. *Investigation on the typing of equality in type systems. (Etude sur le typage de l'égalité dans les systèmes de types)*. PhD thesis, École Polytechnique, Palaiseau, France, 2010.

[SN08] Konrad Slind and Michael Norrish. A brief overview of hol4. In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics*, pages 28–32, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[Son97] Fangmin Song. The expansion postponement in pure type systems. *Journal of Computer Science and Technology*, 12(6):555–563, Nov 1997.

[ST14] Matthieu Sozeau and Nicolas Tabareau. Universe polymorphism in coq. In *International Conference on Interactive Theorem Proving*, pages 499–514. Springer, 2014.

[Ste90] Guy Steele. *Common LISP: the language*. Elsevier, 1990.

[Str92] Thomas Streicher. Independence of the induction principle ad the axiom of choice in the pure calculus of constructions. *Theoretical computer science*, 103(2):395–408, 1992.

[Tas19] Enrico Tassi. Deriving proved equality tests in coq-elpi: Stronger induction principles for containers in coq. In Harrison et al. [HOT19], pages 29:1–29:18.

[Thi18] François Thiré. Sharing a library between proof assistants: Reaching out to the HOL family. In Frédéric Blanqui and Giselle Reis, editors, *Proceedings of the 13th International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, LFMTP@FSCD 2018, Oxford, UK, 7th July 2018*, volume 274 of *EPTCS*, pages 57–71, 2018.

[Tho11] Simon Thompson. *Haskell: the craft of functional programming*, volume 2. Addison-Wesley, 2011.

[TS18] Amin Timany and Matthieu Sozeau. Cumulative inductive types in coq. In Kirchner [Kir18], pages 29:1–29:16.

[Typ05] *Type inference with algebraic universes in the Calculus of Inductive Constructions*, 2005.

[Uni91] *Unification of simply typed lambda-terms as logic programming*, 1991.

[vOvR94] Vincent van Oostrom and Femke van Raamsdonk. Weak orthogonality implies confluence: The higher order case. In Anil Nerode and Yuri V. Matiyasevich, editors, *Logical Foundations of Computer Science, Third International Symposium, LFCS'94, St. Petersburg, Russia, July 11-14, 1994, Proceedings*, volume 813 of *Lecture Notes in Computer Science*, pages 379–392. Springer, 1994.

[Wer94] Benjamin Werner. *Une Théorie des Constructions Inductives*. PhD thesis, Paris Diderot University, France, 1994.

[WR27] Alfred North Whitehead and Bertrand Arthur William Russell. *Principia mathematica; 2nd ed.* Cambridge Univ. Press, Cambridge, 1927.

Index

Nomenclature

$\mathbb{N}_{<n}$	Natural numbers strictly smaller than n
$\mathbb{N}_{\leq n}$	Natural numbers smaller than n
$\equiv_{\Gamma,\Gamma'}$	Extension of $\equiv_{t,t'}$ for typing contexts
$\equiv_{t,t'}$	Equivalence relation on sorts when $t =_\star t'$
$\equiv_{t,t'}^{\preceq_{\mathscr{C}}}$	Free equivalence relation on sorts when $t \preceq_{\mathscr{C}} t'$
$\Gamma =_\star \Gamma'$	Equality of typing contexts modulo the sorts
Γ, Γ'	Concatenation of typing context
$\Gamma \vdash^e_{\mathscr{C}} A \equiv_\beta B : s$	Explicit conversion judgment in semantic CTS
$\preceq_{\mathscr{C}} t,t'$	Free cumulativity relation on sorts when $t \preceq_{\mathscr{C}} t'$
$\Gamma \vdash^e_{\mathscr{C}} A \preceq_{\mathscr{C}} B : s$	Explicit subtyping judgment in semantic CTS
$\Gamma \vdash^e_{\mathscr{C}} t : A$	typing judgment for semantic CTS
$\mathrm{FV}(t)$	the set of free variables in t
EIE_n	Equivalence between CTS and semantic CTS for well-structured derivation trees at level n
EP_n	Expansion postponement for well-structured derivation trees at level n
$\pi \lhd \pi'$	π is a subtree of π'
$\Pi =_\star \Pi'$	Equality of derivation trees modulo the sorts
$\mathscr{C}/_{\equiv}$	Quotient of a CTS specification
$\mathscr{C} \sim^w \mathscr{C}'$	Weak CTS equivalence
$\mathscr{C} \trianglelefteq^w \mathscr{C}'$	Weak CTS embedding
$\mathscr{C}_n$	CTS specification for
$\mathscr{C}_n^C$	CTS specification for Coq
$\mathscr{C}_n^L$	CTS specification for Lean

$\mathscr{C}_n^M$ CTS specification for Matita

$\mathscr{C}_n$ CTS associated to one predicative cumulative hierarchy of universes

$\sigma : \mathscr{C} \to \mathscr{D}$ Specification morphism

$\lhd_{\mathcal{S}_{SC}}$ Ordered relation for ordered specification

C PTS specification for Calculus of Constructions

HOL PTS specification for λHOL

P PTS specification for LF

2 PTS specification for System F

ω PTS specification for System $F\omega$

$\star$ PTS specification for the PTS with a unique sort

$\to$ PTS specification for Simply Typed Lambda Calculus

U PTS specification for System U

$\mathbf{U}^-$ PTS specification for System U^-

$\mathscr{P}_n^A$ PTS specification for Agda

$\mathfrak{0}, \mathfrak{1}, \mathfrak{2}$ Universes in $\mathscr{C}_n$

$t_\star$ sort-erasure of t

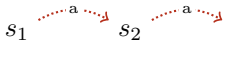
$t =_\star t'$ Equality of terms modulo the sorts

$\mathcal{A}_{\mathscr{C}}$ The set of axioms $\mathcal{A}$ induced by the specification $\mathscr{C}$

Representation of $(s_1, s_2) \in \mathcal{A}_{\mathscr{C}}$

Representation of $(s_1, s_2) \in \mathcal{C}_{\mathscr{C}}$

Representation of $(s_1, s_2, s_2) \in \mathcal{R}_{\mathscr{C}}$



Representation of $(s_1, s_2, s_3) \in \mathcal{R}_{\mathscr{C}}$

$\mathcal{C}^*_{\mathscr{C}}$ Transitive closure of the cumulativity relation

$\mathcal{S}^\top_{\mathscr{C}}$ Set of top sorts (sorts without type).

$\mathcal{T}$ Set of terms

$\sim$ CTS equivalence

$\trianglelefteq$ CTS embedding

$\mathcal{V}$ Set of variables

$\mathcal{C}_{\mathscr{C}}$ The cumulative set $\mathcal{C}$ induced by the specification $\mathscr{C}$

$\uparrow_A^B \mathbf{I}\ t$ A notation for ${}^{s_2}_{s_1}\!\uparrow_A^B \mathbf{I}\ t$

$\uparrow_{s_1}^{s_2} \mathbf{I}\ t$ A notation for ${}^{s'_2}_{s'_1}\!\uparrow^{\mathbf{u}_{s_2,s'_2}\ \mathbf{I}}_{\mathbf{u}_{s_1,s'_1}\ \mathbf{I}} \mathbf{I}\ t$

$\Gamma \vdash_{\mathscr{C}} t \Leftarrow A$ *Check* typing judgment for CTS under specification $\mathscr{C}$

$\Gamma \vdash_{\mathscr{C}} A \overset{?}{\Rightarrow} s$ A is well-sorted and has sort s or $\boldsymbol{s_\infty}$ if $A \in \mathcal{S}^{\top}_{\mathscr{C}}$

$\Gamma \vdash_{\mathscr{C}} t \Rightarrow A$ *Infer* typing judgment for CTS under specification $\mathscr{C}$

$\Gamma \vdash^{a}_{\mathscr{C}} t : A$ Same typing system than $\Gamma \vdash_{\mathscr{C}} t : A$ with a slight change for the application case

$\Gamma \vdash^{t}_{\mathscr{C}r} t : A$ Restriction of CTS with reductions

$\Gamma \vdash_{\mathscr{C}} t : A$ Typing judgment for CTS under specification $\mathscr{C}$

$\Gamma \vdash^{a}_{\mathscr{C}}$ **wf** Same typing system than $\Gamma \vdash_{\mathscr{C}}$ **wf** with a slight change for the application case

$\Gamma \vdash_{\mathscr{R}} t : A$ Typing judgment for PTS modulo under specification $\mathscr{R}$

$\equiv_\alpha$ the relation modulo α

$\equiv$ Transitive, symmetric, reflexive closure stable by syntactic context of a rewriting relation

$\hookrightarrow_\beta$ the relation β-rewrites

$\hookrightarrow_{\beta\delta}$ the relation $\beta\delta$-rewrites

$\hookrightarrow_{\beta\eta}$ the relation $\beta\eta$-rewrites

$\hookrightarrow_\eta$ the relation η-rewrites

$\hookrightarrow^*$ Transitive closure of a rewriting relation

$\hookrightarrow$ A rewriting relation

$\hookrightarrow$ Rewriting relation

$\hookleftarrow$ Inverse of a rewriting relation

$\Gamma\{x \leftarrow t\}$ typing context substitution

$t\{x \leftarrow u\}$ the variable x in t is substituted by u

$\mathscr{C}^{\mathcal{M}}$ minimal specification of $\mathscr{C}$

λS CTS induced by the specification S

WS($\Gamma \vdash_{\mathscr{C}} t : A$) A well-structured judgment

$\mathcal{R}_{\mathscr{C}}$ The set of rules $\mathcal{R}$ induced by the specification $\mathscr{C}$

$\mathscr{C}$ CTS specification

$\mathscr{P}$ PTS modulo specification

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre: Intéropérabilité entre systèmes de preuves avec le cadre logique Dedukti

Mots clés: Dedukti, cadre logique, système de preuve, intéropérabilité, CTS

Résumé: Les systèmes de preuves sont des outils utilisés pour formaliser et prouver des théorèmes. Ces outils sont considérés comme le moyen le plus sûr pour montrer l'absemce de bogues dans les logiciels. Cependant, l'utilisation de ces outils demandent un grand niveau d'expertise ce qui les rend difficile à utiliser. L'intéraction avec un système de preuves demande de prouver et formaliser de nombreux concepts mathématiques. Ce travail particulièrement chronophage requiert la mobilisation d'une force de travail conséquente (par exemple le théorème des quatre couleurs ou le théorème de Hales-Kepler). La diversité des systèmes de preuves induit que ces théorèmes (comme le petit théorème de Fermat) sont prouvés de nombreuses fois. Dans cette thèse, nous étudions tant sur le plan théorique que sur le plan pratique différentes façon de traduire semi-automatiquement des théorèmes prouvés depuis un système de preuve vers un autre.

Title: Interoperability between proof systems using the logical framework Dedukti

Keywords: Dedukti, logical framework, proof system interoperability, CTS

Abstract: Proof systems are tools used to formally prove theorems, and in particular that software is bug-free. Proof systems provide the highest degree of confidence to prove the absence of bugs in software. However, using such tools require a high level of expertise which makes them difficult to use. The interaction with a proof system requires the user to prove and formalize many mathematical concepts. Such work is time-consuming and may require a significant amount of manpower (e.g. four-color theorem or the Hales-Kepler theorem). The diversity of proof systems has the negative consequence that these theorems (e.g. The little Fermat's theorem) are formalized many times. This thesis investigates, both on the theoretical and the practical side, ways to translate (semi-)automatically theorems proved in one proof system to another.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France